1993

A protocol validator for the SCM and CFSM models

Bulbul, Zeki Bulent

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/24220

UNCLASSIFIED

URITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
. DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
. NAME OF PERFORMING ORGANIZATION omputer Science Dept. aval Postgraduate School / 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
. ADDRESS (City, State, and ZIP Code) lonterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
. NAME OF FUNDING/SPONSORING ORGANIZATION / 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION N

. TITLE (Include Security Classification)
PROTOCOL VALIDATOR FOR THE SCM AND CFSM MODELS

. PERSONAL AUTHOR(S)
ulbul Zeki Bulent

a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
laster's Thesis	FROM 09/92 TO 06/93	June 1993	143

. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the fficial policy or position of the Department of Defense or the United States Government.

COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Systems of Communicating Machines, Communicating Finite Sta Machines, SCM, CFSM, Protocol Verification.

. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis introduces and describes a software tool called *Mushroom* which automates the analysis of network protocols spe ed by the Systems of Communicating Machines (SCM) and the Communicating Finite State Machines (CFSM) models. SC a formal model for the specification, verification, and testing of communication protocols. This model was originally deve ed to improve the CFSM model which is a simpler and earlier Formal Description Technique (FDT).

The program is developed as two separate programs in the Ada programming language. The first program automates eith e system state analysis (*Smart Mushroom*), or the full global analysis (*Big Mushroom*) for a protocol specified by the SC odel. The second program called *Simple Mushroom*, automates the global reachability analysis for the CFSM model.

Mushroom greatly facilitates the use of these models for protocol design and analysis. The run time and memory efficienc a previous program was improved to allow the analysis of larger and more complex protocols. The program was also extende accept up to eight machines (processes) in the protocol specification. The user interface of the program has also been in oved.

Mushroom has been used to verify some well known protocols specified by the SCM and CFSM models such as the toke s protocol, Go Back N and Lap-B data link control protocol.

DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
a. NAME OF RESPONSIBLE INDIVIDUAL r. G. M. Lundy	22b. TELEPHONE (Include Area Code) (408) 656-2094/2449 / 22c. OFFICE SYMBOL CS/Ln

FORM 1473, 84 MAR 83 APR edition may be used until exhausted All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

A Protocol Validator for the SCM and CFSM Models

by
Zeki Bulent Bulbul
LTJG, Turkish Navy
B.S., Turkish Naval Academy, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1993

UNCL/
RITY CLASSI
EPOF

ABSTRACT

This thesis introduces and describes a software tool called *Mushroom* which automates the analysis of network protocols specified by the Systems of Communicating Machines (SCM) and the Communicating Finite State Machines (CFSM) models. SCM is a formal model for the specification, verification, and testing of communication protocols. This model was originally developed to improve the CFSM model which is a simpler and earlier Formal Description Technique (FDT).

The program is developed as two separate programs in the Ada programming language. The first program automates either the system state analysis (*Smart Mushroom*), or the full global analysis (*Big Mushroom*) for a protocol specified by the SCM model. The second program called *Simple Mushroom*, automates the global reachability analysis for the CFSM model.

Mushroom greatly facilitates the use of these models for protocol design and analysis. The run time and memory efficiency of a previous program was improved to allow the analysis of larger and more complex protocols. The program was also extended to accept up to eight machines (processes) in the protocol specification. The user interface of the program has also been improved.

Mushroom has been used to verify some well known protocols specified by the SCM and CFSM models such as the token bus protocol, Go Back N and Lap-B data link control protocol.

TABLE OF CONTENTS

I. INTRODUCTION

A. MOTIVATION

In the last decade increasing complexity in computer communication systems have created a growing demand for formal techniques to specify, design, verify and test protocols. In order to have a clear understanding of the protocols, both for the protocol designer and implementor, it is essential to have a formal protocol specification.

There are a large number of formal techniques available for modeling protocols. Most of these methods can be placed into one of the following general classifications [Ref. 1]: communicating finite state machines, Petri nets, programming languages and hybrids. Some models that have found most interest and chosen for standardization are **ESTELLE**, **LOTOS** and **SDL**. Each of these has its own pros and cons.

Systems of communicating machines (SCM) is also a formally defined model for specification, analysis and testing of protocols that is defined in [Ref. 2]. This model uses a combination of finite state machines and variables, which may be local to a single machine or shared by two or more machines, so it can be classified in the models known as "extended finite-state machines." The main goal of the SCM model was to improve the well-known simpler Communicating Finite-State Machines (CFSM) model. The SCM model has been used to specify and analyze several protocols [Ref. 3], [Ref. 4], [Ref. 5], [Ref. 6], [Ref. 7]. Analysis of protocols specified with this model can be executed using a method called *system state analysis*. This analysis is similar to global reachability analysis, but generates a subset of all reachable states. Sometimes this subset is sufficient to verify the protocol. In some cases system state analysis is not sufficient for protocol analysis, and

global analysis is needed. However, it is possible to automate the system state analysis and global analysis based on the SCM model.

Several tools exist for the design and verification of protocols. These tools are very important for increasing the usefulness of the formal description techniques (FDT).

While there is no "perfect" formal specification technique, there is still room for more work to understand the advantages of different formal models and develop better tools to increase the utilization of these models.

B. SCOPE OF THE THESIS

The goal of the thesis is to present a software tool, called **mushroom** that automates the reachability analysis of protocols formally specified using CFSM and SCM models. The name mushroom was chosen as a symbol of something that starts out relatively small (specification) and gets much bigger quickly (analysis). An earlier version of the program [Ref. 8] was capable of generating reachability analysis for the protocols consisting of only two machines. This thesis expands on this earlier work and is capable of analyzing protocols that has any number of machines from two to eight. In addition, the user interface for the program has also been improved. The program was tested against results of several previous works and has confirmed their results. It is also believed that this program will help to solve some problems concerning the SCM model.

C. ORGANIZATION

The thesis has six chapters. Chapter II reviews the Communicating Finite State Machines (CFSM) and Systems of Communicating Machines (SCM) models. In Chapter III, a program called **simple mushroom**, which automates the global reachability analysis based on CFSM model, is described. Chapter IV describes a program that automates the system state analysis (**smart mushroom**), or the full global analysis (**big mushroom**) for

a protocol specified formally using the SCM model. In Chapter V, some examples of the use of the program are given. Chapter VI concludes the thesis with a research review and suggestions for future work.

II. BACKGROUND OF MODELS

A. COMMUNICATING FINITE STATE MACHINES

Communicating finite state machine (CFSM) model is a simple model and perhaps the earliest FDT. In this model, each machine in the network is modeled as a finite automaton or finite state machine (FSM), with communication channels between pairs of machines modeled as one-way, infinite length FIFO queues. There is a great deal of literature on this model [Ref. 9] [Ref. 10] [Ref. 11]. The model is defined for an arbitrary number of machines; however, for simplicity, a two machine model (shown in Figure 1) will be presented here.

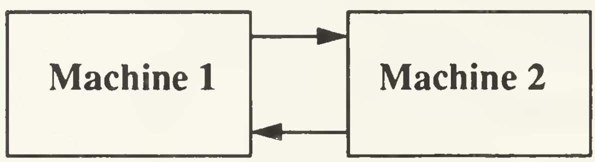

Figure 1: CFSM, 2 machine model representation

1. Model Definition

This section defines the CFSM model [Ref. 12] and provides a simple protocol specification and analysis to clarify the definition.

A *communicating machine* M is a finite, directed labeled graph with two types of edges, *sending* and *receiving*. A sending (receiving) edge is labeled '-*g*' ('+*g*') for some *message* g, taken from a finite set *G* of messages. One of the nodes in *M* is identified as the *initial node*, and each node is reachable from the initial node by some directed path. A node in *M* whose outgoing edges are all sending (receiving) edges is a *sending* (*receiving*) node; otherwise the node is *a mixed* node. If the outgoing edges of each node in *M* have distinct

labels, then M is *deterministic*; otherwise M is *nondeterministic*. The nodes of M are often referred to as *states*; these two terms will be used interchangeably throughout this thesis.

Let M and N be two communicating machines having the same set G of messages; the pair (M,N) is a *network*. A *global state* of this network is a four tuple $[m, c_m, n, c_n]$, where m and n are nodes (states) from M and N, and c_m and c_n are strings from the set G of messages. Intuitively, the global state $[m, c_m, n, c_n]$ means that the machines M and N have reached states m and n, and the communication channels contain the strings c_m and c_n of messages, where c_m denotes the messages sent from M to N in channel C_M, and c_n denotes the messages sent from N to M in channel C_N. In the case of say k number of machines where $k > 2$ the global state can be represented as $[m_1, q_{12}, q_{13}, \ldots, m_2, q_{21}, q_{23}, \ldots, m_3, q_{31}, q_{32}, \ldots, \ldots m_k, q_{k1}, q_{k2}, \ldots]$ where m_is are the nodes of machines M_i and q_{ij} contains the messages sent from M_i to M_j. Subscripts i and j ranges from $1..k$ and $i \neq j$.

The *initial global state* of (M,N) is $[m_0, E, n_0, E]$, where m_0 and n_0 are the initial states of M and N, and E is the empty string.

The network progresses as transitions are taken in either M or N. Each transition consists of a state change in one of the machines, and either the addition of a message to the end of one channel (sending transition) or the deletion of a message from the front of one channel (receiving transition).

A sending transition in M (N) adds a message to the end of channel C_M (C_N); a receiving transition in M (N) removes a message from the front of channel C_N (C_M).

Suppose $+g$ is a receiving transition from state i to j in machine M (N). The transition can be *executed* if and only if M (N) is in state i and the message g is at the front

of the channel C_N (C_M). The execution takes zero time. After its execution, machine M (N) is in state j, and the message g has been removed from the channel C_N (C_M).

Similarly, suppose $-g$ is a sending transition from state i to j in M (N). The transition can be executed if and only if M (N) is in state i. Afterwards, g appears on the end of the outgoing channel, and the machine has transitioned to state j.

Suppose $s_1 = [m,c_i,n,c_j]$ is a global state of (M,N). State s_2 *follows* s_1 if there is a transition (in M or N) which can be executed in s_1 if there is a sequence of states $s_i, s_{i+1}, \ldots, s_{i+p}$ such that s_i follows s_1, s_{i+1} follows s_i, and so on, and s_2 follows s_{i+p}. A state s is *reachable* if it is reachable from the initial state.

The communication of a network(M,N) is *bounded* if, for every reachable state $[m,c_m,n,c_n]$ there is a nonnegative integer k such that $|c_m| \leq k$ and $|c_n| \leq k$, where $|c|$ denotes the number of messages in channel C.

A *reachability graph* of a network (M,N) is a directed graph in which the nodes correspond to the reachable global states of (M,N), and the edges represent the *follows* function. That is, there is an edge from state s_i to state s_j if and only if s_j *follows* s_i. The edges are labeled with the transitions which they represent. This reachability graph can be generated by starting with the initial state, and adding the states which follow it, connecting them to it with edges; and repeating for each new state generated.

The next two definitions are of errors that may occur in a communication protocol, which are detectable by analysis.

A global state $[m,c_m,n,c_n]$ is a *deadlock state* if both m and n are receiving nodes, and $c_m = c_n = E$, where E denotes the empty string.

A global state $[m,c_m,n,c_n]$ is an *unspecified reception state* if one of the following two conditions is true:

(1) m is a receiving state, the message at the head of channel c_n is g, and none of m's outgoing transitions is labeled '+g.'

(2) n is a receiving state, the message at the head of channel c_m is g, and none of n's outgoing transitions is labeled '+g.'

These error conditions can be identified by generating the reachability graph for a network, and inspecting all states as they are generated.

In the next section, an example protocol is specified and analyzed using the CFSM model.

2. An Example of Protocol Specification and Analysis Using CFSM

CFSM specification of an imaginary ring-like network consisting of three communicating machines is shown in Figure 2.

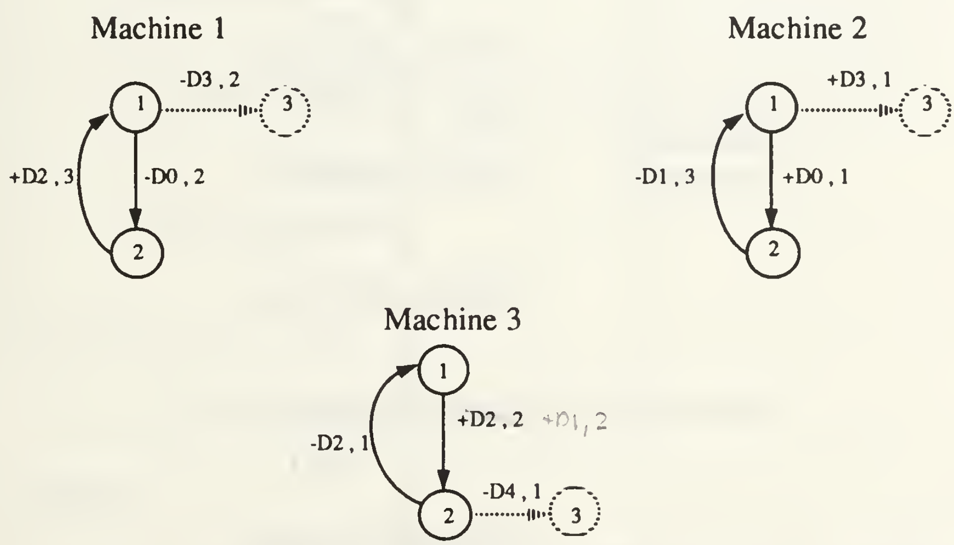

Figure 2: CFSM specification for the example protocol

It is assumed that the protocol is used at the data link layer, making use of the services provided by the physical layer.

Edges are labeled such that the characters following the '-/+' shows the messages and the numbers represent the destination machine. Each machine sends one message to the next machine and receives a message from the previous machine in clockwise direction forming a ring. Ignore the dashed edges and nodes for the time being. The initial state of each machine is 1; thus the initial global state is [1,E,E,1,E,E,1,E,E].

The reachability analysis can be done by a simple procedure. Starting with the initial global state only one transition is possible, the '*-D0*' of the machine 1 from state 1. This leads to global state [2,D0,E,1,E,E,1,E,E]. We can continue the analysis in the same manner detecting the possible transitions from this new global state. The complete reachability analysis is given in Figure 3 consisting of a total of six states.

$$
\begin{tikzcd}
{[1,E,E,1,E,E,1,E,E]} \arrow[d, "-D0,2"] \\
{[2,D0,E,1,E,E,1,E,E]} \arrow[d, "+D0,1"] \\
{[2,E,E,2,E,E,1,E,E]} \arrow[d, "-D1,3"] \\
{[2,E,E,1,E,D1,1,E,E]} \arrow[d, "+D1,2"] \\
{[2,E,E,1,E,E,2,E,E]} \arrow[d, "-D2,1"] \\
{[2,E,E,1,E,E,1,D2,E]} \arrow[uuuuu, bend left=90, "+D2,3"]
\end{tikzcd}
$$

Figure 3: Reachability analysis of the example protocol

In this sample protocol, there are no deadlocks or unspecified receptions. If the dashed edges and states in Figure 2 are added to the specification, the reachability analysis

shown in Figure 4 would be achieved. In this analysis there is one deadlock condition and one unspecified reception. In global state [3,E,E,3,E,E,1,E,E], all the channels are empty and all the nodes are receiving nodes satisfying the deadlock condition. In global state [2,E,E,1,E,E,3,D4,E], machine 1 and machine 2 are in receiving states but none of the outgoing transitions are labeled '*+D4*', satisfying an unspecified reception condition.

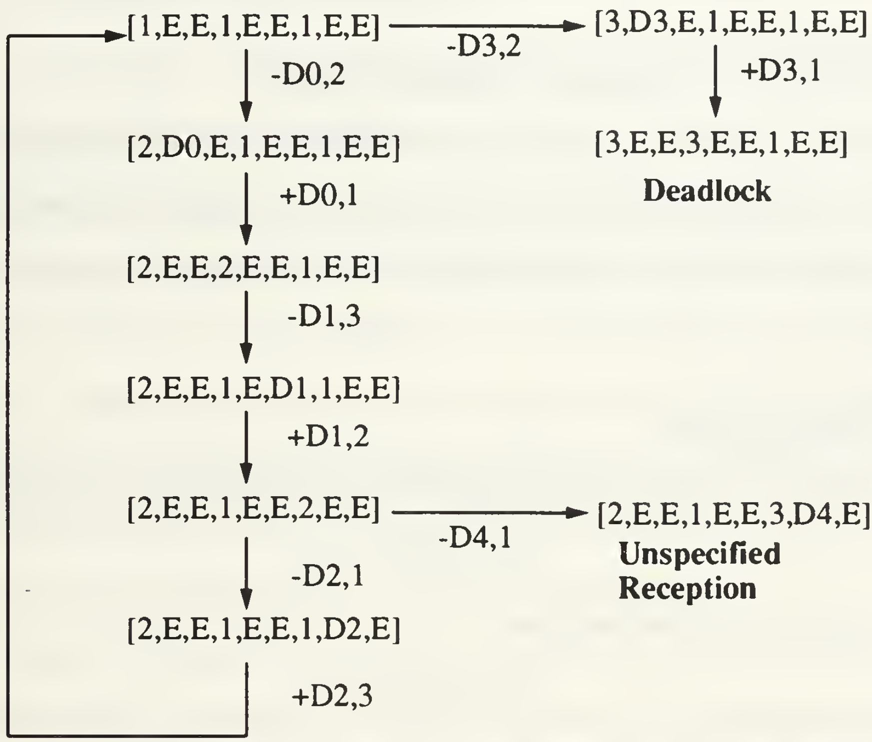

Figure 4: Reachability analysis including errors

3. Summary

The CFSM model is simple and easy to understand. However, as the protocols become more complex, this model becomes difficult to use due to a combinatorial explosion of states. The analysis might not terminate if the queue length is unbounded. The number of states in the reachability graph will be unmanageably large for such complex

protocols even if the queue length is bounded. A computer analysis might eventually terminate, but still the CPU time would be days even months, obviously impractical.

Another disadvantage is that as the protocols become more complex, the specification of the protocol can be so large, consisting of many states and transitions, that it makes it very hard to understand if it is the intended specification. Several examples are given in Chapter V that show the largeness of analysis for some protocols.

B. SYSTEMS OF COMMUNICATING MACHINES

In this section the SCM model is described. First the model definition is given, then the algorithm for generating the system state analysis is described. Finally the model is used for specification and analysis of an example protocol to illustrate the important aspects of the model.

1. Model Definition

A *system of communicating machines* is an ordered pair $C = (M,V)$, where

$$M=\{m_1,m_2,\ldots,m_n\}$$

is a finite set of *machines*, and

$$V=\{v_1,v_2,\ldots,v_k\}$$

is a finite set of shared *variables*, with two designated subsets R_i and W_i specified for each machine m_i. The subset R_i of V is called the set of *read access variables* for machine m_i, and the subset W_i the set of *write access variables* for m_i.

Each machine $m_i \in M$ is defined by a tuple (S_i,s,L_i,N_i,τ_i), where

(1) S_i is a finite set of states;

(2) $s \in S_i$ is a designated state called the *initial state* of m_i;

(3) L_i is a finite set of *local variables*;

(4) N_i is a finite set of names, each of which is associated with a unique pair (p,a), where p is a predicate on the variables $L_i \cup R_i$, and a is an *action* on the variables of $L_i \cup R_i \cup W_i$. Specifically, an action is a partial function

$$a: L_i \text{ X } R_i \rightarrow L_i \text{ X } W_i$$

from the values of the local variables and read access variables to the values of the local variables and write access variables.

(5) τ_i: $S_i \text{ X } N_i \rightarrow S_i$ is a transition function, which is a partial function from the states and names of m_i to the states of m_i.

Machines model the entities, which in a protocol system are processes and channels. The shared variables are the means of communication between the machines. Intuitively, R_i and W_i are the subsets of V to which m_i has read and write access, respectively. A machine is allowed to make a transition from one state to another when the predicate associated with the name for that transition is true. Upon taking the transition, the action associated with that name is executed. The action changes the values of local and/or shared variables, thus allowing other predicates to become true.

The sets of local and shared variables specify a name and range for each. In most cases, the range will be a finite or countable set of values. For proper operation, the initial values of some or all of the variables should be specified.

A *system state tuple* is a tuple of all machine states. That is, if (M,V) is a system of n communicating machines, and s_i, for $1 \leq i \leq n$, is the state of machine m_i, then the n-tuple $(s_1,s_2,...,s_n)$ is the system state tuple of (M,V). A *system state* is a system state tuple, plus the outgoing transitions which are enabled. Thus two system states are *equal* if every machine is in the same state, *and* the same outgoing transitions are enabled.

The *global state* of a system consists of the system state tuple, plus the values of all variables, both local and shared. It may be written as a larger tuple, containing the

system state tuple with the values of the variables. The *initial global state* is the initial system state tuple, with the additional requirement that all variables have their initial values. The *initial system state* is the system state such that every machine is in its initial state, and the outgoing transitions are the same as in the initial global state.

A global state *corresponds* to a system state if every machine is in the same state, and the same outgoing transitions are enabled. Clearly, more than one global state may correspond to the same system state.

Let $\tau\ (s_1,n) = s_2$ be a transition which is defined on machine m_i. Transition τ is *enabled* if the enabling predicate p, associated with name n, is true. Transition τ may be *enabled* whenever m_i is in state s_i and the predicate p is true (enabled). The *execution* of τ is an atomic action, in which both the state change and the action a associated with n occur simultaneously.

It is assumed that if a transition is enabled indefinitely, then it will eventually occur. This is an assumption of *fairness*, and is needed for the proofs of certain properties.

2. Algorithm: System State Analysis

The process of generating the set of all system states reachable from the initial state is called *system state analysis*. This analysis constructs a graph, whose nodes are the reachable system states, and whose arcs indicate the transitions leading from each system state to another. This graph may be generated by a mechanical procedure which consists of the following three steps [Ref. 1]:

1. Set each machine to its initial state, and all variables to their initial values. The initial set of reachable system states consists of only the initial system state; the initial graph is a single node representing this state.

2. From the current system state vector and variable values, determine which transitions are enabled. For each of these transitions, determine the system state which results from its execution. *If* this state (with the same enabled transitions)

has already been generated, then draw an arc from the current state to it, labelling the arc with the transition name. *Otherwise*, add the new system state to the graph, draw an arc from the current state to it, and label the arc with the name of the transition.

3. For each new state generated in step 2, repeat step 2. Continue until step 2 has been repeated for each system state thus generated, and no more new states are generated.

3. An Example of Protocol Specification and Analysis Using SCM

The specification of an imaginary ring-like network consisting of three machines similar to the CFSM example in the previous section is given in Figure 5. The specification consists of the finite state machines, the local and shared variables, and the predicate action table, shown in Table 1. The local variables are: *in_buff1*, *in_buff2*, *in_buff3*, *out_buff1*, *out_buff2*, and *out_buff3* and shown under the corresponding FSMs with their initial values. The shared variables are: *CHAN1*, *CHAN2*, and *CHAN3* and shown between the two machines. The initial state of each machine is 0, with the shared variables and local variables are empty except the local variable **out_buff1**, which has data in it. **E** in the predicate-action table shows the empty string. A character **D** will be used to represent the data in the **out_buff1** local variable. Other notations in the predicate-action table are intuitive.

Each machine sends one message to the next machine and receives a message from the previous machine in clockwise direction forming a ring. The global reachability analysis, shown in Figure 6, has 12 states. The system state analysis, shown in Figure 7, has only 6 states. The subscripts in Figure 7 are used so that distinct system states having the same tuple (but not the outgoing transitions) may easily distinguished.

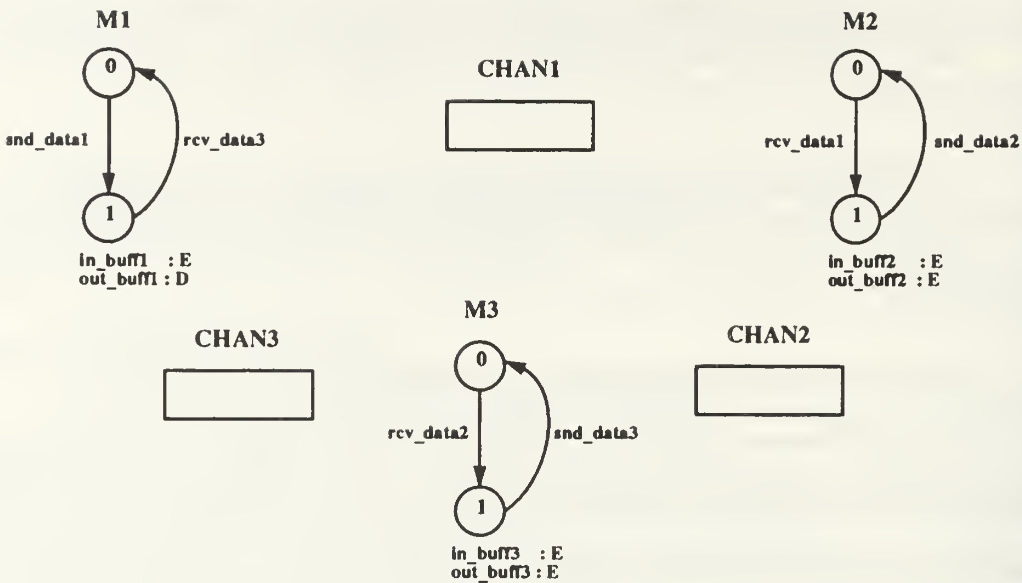

Figure 5: FSMs and variables for the example protocol

TABLE 1: PREDICATE-ACTION TABLE FOR THE EXAMPLE PROTOCOL

Transition	Enabling Predicate	Action
snd_data1	CHAN1 = E ∧ out_buff1 ≠ E	CHAN1 ← out_buff1 out_buff1 ← E
rcv_data3	CHAN3 ≠ E	in_buff1 ← CHAN3 out_buff1 ← in_buff1 CHAN3 ← E
snd_data2	CHAN2 = E ∧ out_buff2 ≠ E	CHAN2 ← out_buff2 out_buff2 ← E
rcv_data1	CHAN1 ≠ E	in_buff2 ← CHAN1 out_buff2 ← in_buff2 CHAN1 ← E
snd_data3	CHAN3= E ∧ out_buff3 ≠ E	CHAN3 ← out_buff3 out_buff3 ← E
rcv_data2	CHAN2 ≠ E	in_buff3 ← CHAN2 out_buff3 ← in_buff3 CHAN2 ← E

[m1,in_buff1,out_buff1,m2,in_buff2,out_buff2,m3,in_buff3,out_buff3,CHAN1,CHAN2,CHAN3]

[0,E,D,0,E,E,0,E,E,E,E,E]
↓ snd_data1
[1,E,E,0,E,E,0,E,E,D,E,E]
↓ rcv_data1
[1,E,E,1,D,D,0,E,E,E,E,E]
↓ snd_data2
[1,E,E,0,D,E,0,E,E,E,D,E]
↓ rcv_data2
[1,E,E,0,D,E,1,D,D,E,E,E]
↓ snd_data3
[1,E,E,0,D,E,0,D,E,E,E,D]
↓ rcv_data3
[0,D,D,0,D,E,0,D,E,E,E,E]
↓ snd_data1
[1,D,E,0,D,E,0,D,E,D,E,E]
↓ rcv_data1
[1,D,E,1,D,D,0,D,E,E,E,E]
↓ snd_data2
[1,D,E,0,D,E,0,D,E,E,D,E]
↓ rcv_data2
[1,D,E,0,D,E,1,D,D,E,E,E]
↓ snd_data3
[1,D,E,0,D,E,0,D,E,E,E,D]

rcv_data3

Figure 6: Global reachability analysis for the example protocol

Thus, for this protocol we have 6 system states, and 12 global states. For more complex protocols, the difference between these numbers can be much more. For example, a sliding window protocol with a window size of 8 the system state analysis was shown to generate 165 states, while the full global analysis generated 11880 states [Ref. 1].

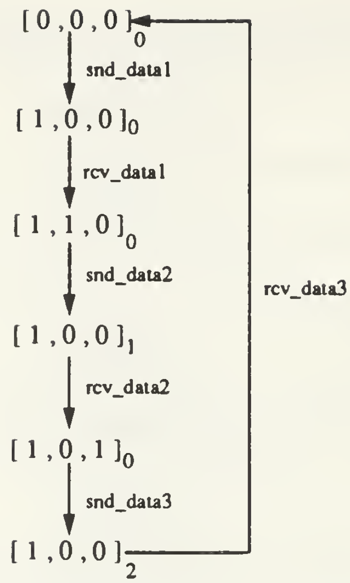

Figure 7: System state analysis for the example protocol

4. Summary

The SCM model has desirable properties which overcome some of the disadvantages of the CFSM model. One of the advantages of the SCM model is that it greatly reduces the number of state explosion through the use of system state analysis. In some cases, however, the system state analysis is not sufficient for protocol analysis, and some other method - such as global analysis - must be done. A problem with the system state analysis is the loops in the state machines which may cause an insufficient analysis. This problem is illustrated with an example in Chapter V.

Another advantage of SCM model is that it allows communication between machines in nonsequential manner, unlike a FIFO queue representation in the CFSM model. The SCM model specification is also easier to understand than the CFSM model for more complex protocols.

III. SIMPLE MUSHROOM: A PROGRAM FOR AUTOMATING CFSM REACHABILITY ANALYSIS

This Chapter and the next Chapter will describe a program called **mushroom**, which was written in the Ada programming language. **Mushroom** automates the reachability analysis of protocols specified by the CFSM and the SCM models. The Mushroom program was first developed as two separate programs. The first program called **simple mushroom,** automates the CFSM analysis. The second program automates either system state analysis (**smart mushroom**), or the full global analysis (**big mushroom**) for a protocol specified formally by the SCM model. The General structure of the Mushroom program is shown in Figure 8.

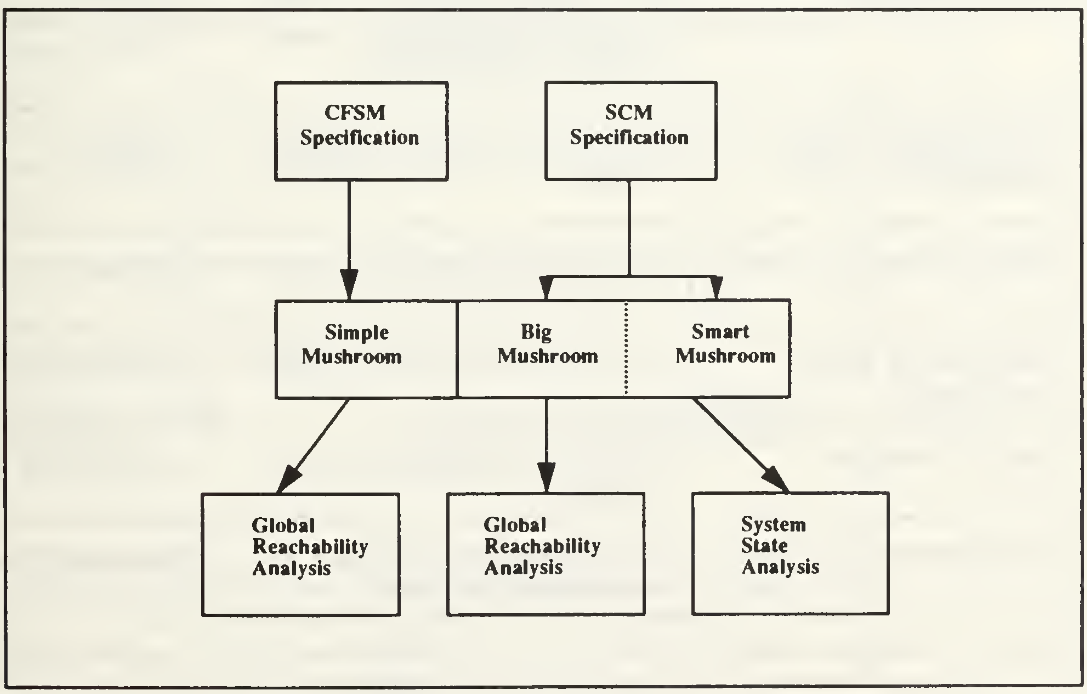

Figure 8: General structure of Mushroom program

The *Simple Mushroom* program, is described in this chapter in four sections: program structure, inputs to the program, generating the reachability analysis, and outputs of the program.

A. PROGRAM STRUCTURE

The *Simple Mushroom* program consists of Ada subprograms (procedures and functions), which are separate compilation units and subunits of compilation units. Related subprograms are also gathered in the same files. The compilation units of the program are shown in Table 2. Procedure **main** is the *parent unit.* All of the subprograms are the subunits of procedure main. [Ref. 13]

TABLE 2: SIMPLE MUSHROOM PROGRAM COMPILATION UNITS

Compilation Unit	Description	File name
main (procedure)	This is the *parent unit.* Contains the main data structures, global variables, and the driver.	tmain.a
load_machine_array (procedure)	Builds the adjacency lists from FSMs.	tinput.a
read_in_file (procedure)	Parses the input FSM text file.	tinput.a
build_Gstate_graph (procedure)	Generates the reachability graph.	treachability.a
IsEqual (function)	Compares two global states for equality.	treachability.a
hash (function)	Generates an index number according to the hashing function.	treachability.a
clear_pointers (procedure)	Deallocates the dynamic memory space for another analysis.	treachability.a
find_tuple (function)	Searches the reachability graph for the equivalent tuples using external (open) hashing.	tsearch.a

Compilation Unit	Description	File Name
clear_hash_array (procedure)	Clears the hash array and deallocates the memory.	tsearch.a
Print Queue (procedure)	Prints the FIFO queues.	toutput.a
output_Gstate_transition (procedure)	Outputs the transition name.	toutput.a
output_Gstate_node (procedure)	Outputs the machine states, unspecified receptions, and the states with deadlocks.	toutput.a
output_machine_arrays (procedure)	Outputs the FSM description in a tabular format.	toutput.a
output_unexecuted_transitions (procedure)	Outputs the unexecuted transitions.	toutput.a
create_output_file (procedure)	Creates an output file for storing the analysis results.	toutput.a
output_analysis (procedure)	Driver for the output subprograms.	toutput.a
system_call (procedure)	Interface procedure for Unix system calls via C.	tsystem.a
message_queues (package)	Implements the queue operations for the FIFO communication channels.	tqueues.a
pointer_queues (generic package)	Implements the queue operations for the pointer queue that stores the globals tuples temporarily.	tqueues_2.a

The method of splitting the program into separate compilation units has permitted a hierarchical program development.

B. INPUT

The CFSM specification of a protocol consists of only FSMs of the communicating machines. In the program, FSMs are represented with a text file. The user enters the directed graphs as a text file using some reserved words, numbers, and characters representing the machines, states and the transitions. The list of reserved words and the syntax for the FSM text description are shown in Figure 9 in Backus-Naur Form (BNF).

```
reserved_word ::= start
                | number_of_machines
                | machine
                | state
                | trans
                | initial_state
                | finish
number_of_machines  <machine_number>
machine  1 | <machine_number>
state  <state_number>
trans { + | - } <message> <next_state> <next_machine>
initial_state  <state_number> <state_number> [<state_number>] [<state_number>]
        [<state_number>] [<state_number>] [<state_number>] [<state_number>]
<machine_number> ::= 2|3|4|5|6|7|8
<state_number> ::= 0|2|3|.....|50
<message> ::= { <letter> | <digit> } [{ <letter> | <digit> }] [{ <letter> | <digit> }]
<next_state> ::= <state_number>
<next_machine> ::= 1| <machine_number>
<letter> ::= a|b|...|z|A|B|...|Z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

Figure 9: Syntax for the text description of FSM

As can be seen from Figure 9, the maximum number of machines allowed is eight, and the number of states for each machine can be from 0 to 50. Transition names must be at most three characters long and may be any combination of letters or digits. These constraints can be relaxed with slight modifications to the program, if necessary.

The input file for the example protocol in Chapter II for the CFSM model is shown in Figure 10. For example, "trans -D3 3 2" represents a transition from state 1 to state 3 (first number) in machine 1 sending ("-" sign) the message "D3" to machine 2. "Initial_state 1 1 1" means that the initial states of machine 1, machine 2, and machine 3 are state 1.

```
start
number_of_machines 3
machine 1
state 1
trans -D3 3 2
trans -D0 2 2
state 2
trans +D2 1 3
machine 2
state 1
trans +D3 3 1
trans +D0 2 1
state 2
trans -D1 1 3
machine 3
state 1
trans +D2 2 2
state 2
trans -D4 3 1
trans -D2 1 1
initial_state 1 1 1
finish
```

Figure 10: Text file description of the FSM

First, this file is parsed by read_in_file procedure and tokens are generated. Then, Load_machine_array procedure constructs an adjacency list which represents the FSMs.

The data structure for the adjacency list is shown below:

```
type cfsm_transition_type is (s,r,u);
type visit_type is (yes,no);
type state_type is range 0..50;
type next_machine_type is range 1..8;
type machine_array_record_type;
type Slink_tupe is access machine_array_record_type;
type machine_array_record_type is
  record
    transition      : cfsm_transition_type := u;
    message         : message_queue.message_queue_type;
    next_Mstate     : state_type := 0;
    other_machine   : next_machine_type := 1;
    visited         : visit_type := no;
    Slink           : Slink_type := null;
  end record;
type machine_array_type is array(state_type range 0..50) of Slink_type;
type system_array_type is array(next_machine_type range 1..8) of machine_array_type;
```

The adjacency list for the example protocol is depicted in Figure 12. This adjacency list is used for constructing the global reachability graph. The adjacency list contains all the necessary information for generating the global reachability graph.

The user also provides the name of the text input file and a file name for storing the analysis results. Input file name must end with ".fsm" extension to prevent confusion. The output file name must be no more than 20 characters long.

C. REACHABILITY ANALYSIS

After reading the input file the program starts generating the global reachability graph. The program uses the adjacency list and the initial state to construct the global reachability graph. Starting with the initial state, the new states are added and linked to the graph dynamically. The algorithm to construct the global reachability graph is given in Figure 13.

During the graph construction, the program also detects the global states with deadlocks and unspecified receptions. The program also finds the maximum message queue size and channel overflows. Analysis results are stored in the output file in parallel

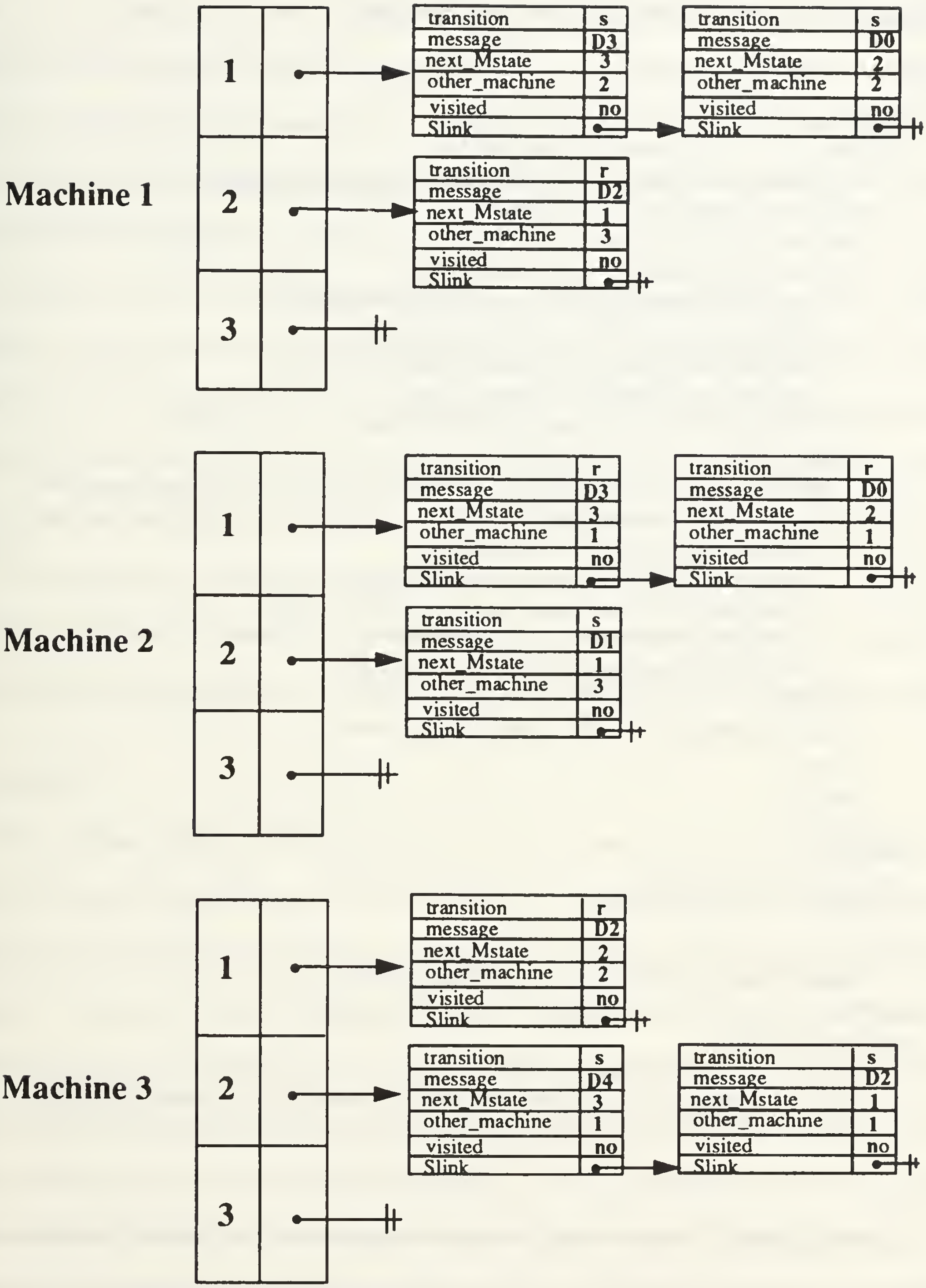

Figure 12: Adjacency list for the example ring protocol in Chapter II

with the graph construction. This prevents the traversal of the entire graph one more time at the end of the program and decreases the run time.

```
loop (main loop)
 for index1 in 1 .. total_number_of_machines loop
   place_holder(index1) := machine_array(index1) (M_state(index1))
   while (place_holder(index) /= null) loop
     loop
       if (place_holder(index1).transition = s) then
         Enqueue the message into the corresponding message queue
         search the graph for this new global state tuple
         if not found then create a new node and link to the graph
           Enqueue this new node to the pointer_queue
         else link the transition to found global state tuple
       else
         if(place_holder(index1).transition) = r and at least one of the message queues for
         this machine is not empty then
           find this message queue and Dequeue
           search the graph for this new global state tuple
           if not found then create a new node and link to the graph
             Enqueue this new node to the pointer_queue
           else link the transition to found global state tuple
        end if;
        place_holder(index1) := place_holder(index1).Slink
        exit
     end loop
   end loop
 end loop
 if pointer_queue empty then
   exit
 else
   Dequeue pointer_queue and update M_state for this new node
 end if
end loop (main loop)
```

Figure 13: Algorithm for generating global reachability graph for CFSM

One of the most time consuming procedures is the search algorithm for detecting if a node was previously created. The previous version of the program [Ref. 8] used a *depth first search / breadth first search* in a recursive manner. In this program, the search is made

more efficient using a *hashing* algorithm. The *hash function* is obtained from the machine states of the global tuple which has provided an efficient mapping. Therefore, the complexity of the search algorithm is $O(1)$ when the hash function generates a distinct index (no collision) and $O(n)$ when the same index is generated, where n is the number of hash collisions for that state. In many sample runs of the program, the complexity was $O(1)$ for about 30% of the global states, and 3 nodes had to be traversed on the average for 70% of the global states. The reachability analysis is limited by the storage capacity of the computer. The run time is also another factor that must be considered. The largest analysis carried out by the program thus far has generated about 160,000 states in 12 hours for a six machine protocol specification. Some alternative methods for improving the efficiency of the program and analysis size using other search techniques are discussed in Chapter VI.

The structure of a global node is shown in Figure 14. The maximum number of outgoing transitions is limited to 7, which can be increased if needed. Also, a maximum channel capacity of 6 messages is introduced to ensure that the analysis eventually stops.

D. OUTPUT

The program stores the analysis results in a file named by the user during the reachability graph construction. This file contains the specification in a tabular format, reachability graph and the results of the analysis consisting of the number of states generated, number of states analyzed, number of deadlocks, number of unspecified receptions, maximum message queue size and number of channel overflows. Global states with deadlocks and unspecified receptions are also marked in the reachability graph. The output file also lists the unexecuted transitions. A menu is displayed at the end of the analysis. From this menu the user has the option of displaying or printing the results or continuing the program for another analysis.

If the analysis generates more than 2000 states, the program gives an interim summary of the analysis and asks the user if they would like to continue. If the user wishes to continue, analysis proceeds in steps of 1000 states until the analysis ends or the user terminates the analysis (as long as memory is available). For analyzing large protocols, the number of states between these “stops” can be made larger (for example, increments of 5000 or 10000). The program output for the example protocol in Chapter II is given in Figure 15.

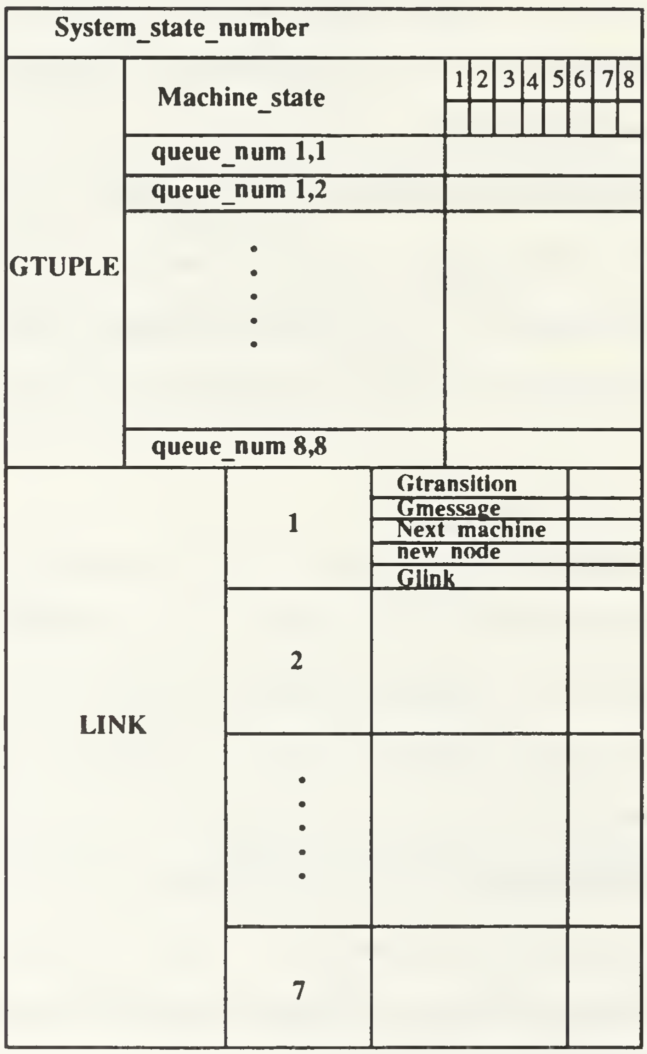

Figure 14: Global state structure with outgoing transitions

```
              REACHABILITY ANALYSIS of : ring.fsm
                       SPECIFICATION
      ---------------------------------------------------
      |       Machine  1  State Transitions             |
      ---------------------------------------------------
      | From   |  To   | other machine | Transition    |
      ---------------------------------------------------
      |   1    |  2    |       2       |   s  d0       |
      |   1    |  3    |       2       |   s  d3       |
      |   2    |  1    |       3       |   r  d2       |
      ---------------------------------------------------
      ---------------------------------------------------
      |      Machine  2  State Transitions              |
      ---------------------------------------------------
      | From   |  To   | other machine | Transition    |
      ---------------------------------------------------
      |   1    |  2    |       1       |   r  d0       |
      |   1    |  3    |       1       |   r  d3       |
      |   2    |  1    |       3       |   s  d1       |
      ---------------------------------------------------
      ---------------------------------------------------
      |      Machine  3  State Transitions              |
      ---------------------------------------------------
      | From   |  To   | other machine | Transition    |
      ---------------------------------------------------
      |   1    |  2    |       2       |   r  d1       |
      |   2    |  1    |       1       |   s  d2       |
      |   2    |  3    |       1       |   s  d4       |
      ---------------------------------------------------
             REACHABILITY GRAPH
  1 [ 1,E,E, 1,E,E, 1,E,E]
      -d0  2  [ 2,d0,E,1,E,E,1,E,E]   2
      -d3  2  [ 3,d3,E,1,E,E,1,E,E]   3
  2 [ 2,d0 ,E, 1,E,E,1,E,E]
      +d0  1  [ 2,E,E,2,E,E,1,E,E]    4
  3 [ 3,d3,E,1,E,E,1,E,E]
      +d3  1  [ 3,E,E,3,E,E,1,E,E]    5
  4 [ 2,E,E,2,E,E,1,E,E]
      -d1  3  [ 2,E,E,1,E,d1,1,E,E]   6
  5 [3,E,E,3,E,E,1,E,E]**********DEADLOCK condition*************
  6 [ 2,E,E,1,E,d1,1,E,E]
      +d1  2  [ 2,E,E,1,E,E,2,E,E]    7
  7 [ 2,E,E,1,E,E,2,E,E]
      -d2  1  [ 2,E,E,1,E,E,1,d2,E]   8
      -d4  1  [ 2,E,E,1,E,E,3,d4,E]  9
  8 [ 2,E,E,1,E,E,1,d2,E]
      +d2  3  [ 1,E,E,1,E,E,1,E,E]    1
  9[2,E,E,1,E,E,3,d4,E]**********Unspecified Reception**********

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
----------------------------------------------------
Total number of states generated : 9
Number of states analyzed : 9
Number of deadlocks : 1
Number of unspecified receptions : 1
Maximum message queue size :  1
Channel overflow :NONE
                          UNEXECUTED TRANSITIONS
                            *****NONE*****
```

Figure 15: Program output for the example ring protocol

IV. SMART AND BIG MUSHROOM: A PROGRAM FOR AUTOMATING SCM REACHABILITY ANALYSIS

In this Chapter, programs that automate either system state analysis (**smart mushroom**), or the full global analysis (**big mushroom**) for a protocol specified by SCM are described. The program is described in four sections: general program structure, inputs to the program, generating the reachability graph, and outputs of the program.

A. PROGRAM STRUCTURE

Program structure of *Smart Mushroom* and *Big Mushroom* are similar to the structure of *Simple Mushroom.* The SCM model specification is more complicated than the CFSM specification, but this complexity in the specification brings some advantages to the analysis as mentioned in Chapter II. A protocol specified by the SCM model consists of FSMs, variable definitions, and predicate-action table, rather than just the FSMs as in CFSM model.

FSMs are entered into the program in the same manner as in *Simple Mushroom* program using a text file. The variable definitions and predicate-action table must also be entered into the program. The user enters these parts by completing Ada packages[1] and subprograms using the templates provided.

The compilation units for the program are shown in Table 3. The user has access to the last four packages/subprograms. Once the user completes these subprograms using the templates and compiles them with the other compilation units, the analysis of the specified

1. Ada packages are one of the four forms of program unit, of which programs can be composed. The other forms are subprograms, task units, and generic units. Packages allow the specification of groups of logically related entities. In their simplest form packages specify pools of common object and type declarations. [Ref. 13]

protocol can be performed. Construction of the specification in the form of Ada packages and subprograms is explained in the next section.

TABLE 3: SMART AND BIG MUSHROOM PROGRAM COMPILATION UNITS

Compilation Unit	Description	File name
Main (procedure)	This is the *parent unit*. Contains the main data structures, global variables, and the driver.	smain.a
load_machine_array (procedure)	Builds the adjacency lists from FSMs.	sinput.a
read_in_file (procedure)	Parses the input FSM text file.	sinput.a
build_Gstate_graph (procedure)	Generates the global reachability graph.	sg_reachability.a
build_system_state_graph (procedure)	Generates the system reachability graph.	sg_reachability.a
hash (function)	Generates an index number according to the hashing function.	sg_reachability.a
clear_pointers (procedure)	Deallocates the dynamic memory space for another analysis.	sg_reachability.a
search_for_Gtuple (function)	Searches the reachability graph for the equivalent global tuples using hashing.	sg_search.a
clear_hash_array (procedure)	Clears the hash array and deallocates the memory for global reachability analysis.	sg_search.a
search_for_Stuple (function)	Searchs the reachability graph for the equivalent system tuples using hashing.	sg_search.a
clear_hs_hash_array (procedure)	clears the hash array and deallocates the memory for system state analysis.	sg_search.a
output_Gstate_node (procedure)	Outputs the machine states, and states with deadlock for global reachability analysis.	sg_output.a

Compilation Unit	Description	File Name
output_sys_node (procedure)	Outputs machine states, and states with deadlock for system state analysis.	sg_output.a
output_Gstate_transition (procedure)	Outputs the transition name for global reachability analysis.	sg_output.a
output_sys_transition (procedure)	Outputs the transition name for system state analysis.	sg_output.a
output_unexecuted_transitions (procedure)	Outputs the unexecuted transitions.	sg_output.a
output_machine_arrays (procedure)	Outputs the FSM description in a tabular format.	sg_output.a
output_analysis (procedure)	Driver for the output subprograms.	sg_output.a
system_call (procedure)	Interface program for Unix system calls via C.	ssystem.a
queues (generic package)	Implements the queue operations for the pointer queue that stores the nodes temporarily.	squeues.a
stacks (generic package)	Implements the stack operations for storing enabled transitions.	sstacks.a
definitions (package)	Includes user defined local and shared variables.	named by the user
Analyze_Predicates (procedure) there is one for each machine	Determines the enabled transitions from the predicates.	named by the user
Action (procedure)	Executes the actions for the enabled transitions.	named by the user
output_gtuple (procedure)	Outputs the global state tuples in a format defined by the user.	named by the user

B. INPUT

The inputs to the program consists of three parts, as mentioned earlier. FSMs are entered using a text file representation as in *Simple Mushroom* program. Variables and predicate-action table are entered as Ada packages/subprograms. The user needs to complete these packages and subprograms by filling in templates provided.

The Ada package template for the variable declarations is called "**definitions.**" The predicate-action table is entered using an Ada subprogram template which consists of one procedure named "**Action**" and two to eight procedures called "**Analyze_Predicates_Machine***" according to the number of machines in the protocol. The "*" at the end of the procedure name is replaced by the corresponding machine number for each machine in the protocol.

After completing the templates described above, the user must compile these units with the other compilation units listed in Table 3. The program units can be compiled by entering a "make" command. The "make" command executes a list of shell commands in the "Makefile" file which contains the commands for compiling the program units according to their dependencies. After issuing the "make" command, the executable file is stored in a file named "scm." The "Makefile" is provided to the user with the mushroom program.

Each of these program units will be explained in the following subsections. The example ring protocol described in Chapter II is also used to illustrate how to complete the templates.

1. Finite State Machines

There are a few differences in the FSM description of *Smart* and *Big Mushroom* programs from *Simple Mushroom* program. The same reserved words are used to write the

FSM text file. These are listed in Figure 9. The syntax changes that must be made to this form are shown in Figure 16.

In the SCM model, explicit machine numbers to show which machine the message sent to or received from are not needed for the transition names. Since shared variables are used for communication between machines, this information is included in the predicate-action table. The FSM text file for the example ring protocol is shown in Figure 17.

trans <transition name> <next_state>
<transition name> ::= <identifier>
<identifier> ::= {[underline] | letter_or_digit}
<letter_or_digit> ::= <letter > | <digit>

Figure 16: Syntax changes for FSM description of SCM model

```
start
number_of_machines 3
machine 1
state 0
trans snd_data1 1
state 1
trans rcv_data3 0
machine 2
state 0
trans rcv_data1 1
state 1
trans snd_data2 0
machine 3
state 0
trans rcv_data2 1
state 1
trans snd_data3 0
initial_state 0 0 0
finish
```

Figure 17: Text file description of the example ring protocol

The FSM text file is read by the input procedures and the adjacency list, which is used during the construction of system and global reachability graphs is generated. The data structure for the adjacency list is shown in Figure 18.

```
visit_type is (yes, no);
type machine_array_record_type;
type Slink_type is access machine_array_record_type;
type machine_array_record_type is
  record
    transition      : scm_transition_type := unused;
    next_Mstate     : natural := 0;
    visited         : visit_type := no;
    Slink           : Slink_type := null;
  end record;
type machine_array_type is array(integer range 0 .. 50) of Slink_type;
type system_array_type is array (1 .. num_of_machine) of machine_array_type;
```

Figure 18: Data structure for the adjacency list.

2. Variable Definitions

The user defines the protocol variables in an Ada package named *definitions*. This package includes the local variables for each machine and the global variables, which are considered shared and allow communication between machines. A variable can be one of the Ada defined types such as: integer, array, string, record, character, boolean, etc. These types and their subtypes are used to define the protocol variables.

The template for the *definitions* package is given in Figure 19. The shaded areas show where the variables of the protocol are inserted by the user. Additional type declarations should be placed before the machine type declarations.

The variable declarations for the example ring protocol is also shown in Figure 20. The local variables of the protocol are: **in_buff1**, **in_buff2**, **in_buff3**, **out_buff1**, **out_buff2**, and **out_buff3.** The shared variables are: **CHAN1**, **CHAN2** and **CHAN3**. The type definition, *Dummy_type* is placed in each of the local variable declarations of

machines in case the protocol has less than eight machines. When declaring the local variables for each machine, this dummy variable can be deleted from the corresponding machine. The initial values of the variables are also assigned with the variable declarations.

```
with TEXT_IO;
use TEXT_IO;
package definitions is                          Number of machines in the specification
  num_of_machines : constant := [redacted] ;     (can be 2 to 8)
  type scm_transition_type is ( [redacted] );
  type dummy_type is range 1..255;

  type machine1_state_type is                    Transition names of FSMs
    record
      dummy : dummy_type;
      [redacted]
    end record;
.
.
.
  type machine8_state_type is
    record                                       Local variables for machines 1 to 8
      dummy : dummy_type;
      [redacted]
    end record;
  type global_variable_type is
    record
      [redacted]                                 Global (shared) variables
    end record;
end definitions;
```

Figure 19: Template for *definitions* package

3. Predicate-Action Table

The predicate-action table is represented by a number of subprograms as separate compilation units. These subprograms are named *Analyze_Predicates* and are used to determine the enabled transitions for each machine. The procedure named *Action* executes the actions to be taken for the corresponding enabled predicates. There is one

Analyze_Predicates procedure for each machine and one *Action* procedure for the protocol.

The template for the *Analyze_Predicates* procedure is shown in Figure 21.

```
with TEXT_IO;
use TEXT_IO;
package definitions is
     num_of_machines : constant := 3;
     type scm_transition_type is (snd_data1,rcv_data3,snd_data2,
                                  rcv_data1,snd_data3,rcv_data2,unused);
  type buffer_type  is (D,E );
  package buff_enum_io is new enumeration_io (buffer_type);
  use buff_enum_io;
  type dummy_type is range 1..255;

  type machine1_state_type is
    record
       out_buff1 : buffer_type := D;
       in_buff1 : buffer_type:= E;
    end record;
  type machine2_state_type is
    record
       out_buff2,
       in_buff2  : buffer_type:= E;
    end record;
  type machine3_state_type is
    record
       out_buff3,
       in_buff3  : buffer_type := E;
    end record;
  type machine4_state_type is
    record
       dummy : dummy_type;
    end record;
                •
                •
                •
  type machine8_state_type is
   record
     dummy : dummy_type;
   end record;
  type global_variable_type is
    record
       CHAN1,
       CHAN2,
       CHAN3  : buffer_type := E;
    end record;

end definitions;
```

Figure 20: Completed *Definitions* package for the example ring protocol

```
separate(main)
procedure Analyze_Predicates_machine1(local : machine1_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is
begin
  case s is                                          Enabling condition
    when 0 =>
      if ( [redacted] ) then
        push(w, [redacted] );
      end if;

    when 1 =>
        .                                            Enabled transition
        .
        .
    when others =>
        null;
  end case;
end Analyze_Predicates_machine1;
```

Figure 21: Template for *Analyze_Predicates* procedures

The user completes the template for each state of the machines. For each machine state there is one "when" statement. "If" statements specify the predicates for possible transitions from the current state. The "Push" statement stores these transitions in the stack. Since more than one transition can be enabled in some states, a stack is used to store all possible transitions. The "s" parameter, in the formal parameter list of the procedure, passes the machine state; and the "w" parameter passes the stack name to the procedure. The file for the example ring protocol is given in Figure 22.

The template for the *Action* procedure is shown in Figure 23. The enabled transitions are passed into this procedure through the "in_transition" formal parameter and the necessary changes are made to the local and shared variables by the *Action* procedure. The "out_system_state" parameter passes the changed protocol variables to the calling procedure. The completed *Action* procedure is shown in Figure 24. Text in boldface shows the user defined parts.

```
separate (main)
procedure  Analyze_Predicates_Machine1(local : machine1_state_type;  GLOBAL: global_variable_type;
                                                   s : natural;  w : in out transition_stack_package.stack) is
 begin
  case s is
   when 0 =>
    if( (GLOBAL.CHAN1 = E) and ( LOCAL.out_buff1 /= E) ) then
      Push(w,snd_data1);
    end if;
   when 1 =>
    if (GLOBAL.CHAN3 /= E)  then
      Push(w,rcv_data3);
    end if;
   when others =>
    null;
  end case;
end Analyze_Predicates_Machine1;
separate (main)
procedure Analyze_Predicates_Machine2(local : machine2_state_type; GLOBAL: global_variable_type;
                                                   s: natural; w : in out transition_stack_package.stack) is
begin
  case s is
   when 0 =>
     if (GLOBAL.CHAN1 /= E) then
      Push(w,rcv_data1);
    end if;
   when 1 =>
     if ( (GLOBAL.CHAN2 = E) and (local.out_buff2 /= E) ) then
      Push(w,snd_data2);
     end if;
   when others =>
    null;
  end case;
end Analyze_Predicates_Machine2;
separate (main)
procedure Analyze_Predicates_Machine3(local : machine3_state_type; GLOBAL: global_variable_type;
                                                   s : natural; w : in out transition_stack_package.stack) is
begin
  case s is
   when 0 =>
    if ( GLOBAL.CHAN2 /= E ) then
     push(w,rcv_data2);
    end if;
   when 1 =>
   if ( (GLOBAL.CHAN3 = E ) and (local.out_buff3 /= E ) ) then
    push(w,snd_data3);
   end if;
   when others =>
    null;
  end case;
end Analyze_Predicates_Machine3;
separate (main)
procedure Analyze_Predicates_Machine4(local :machine4_state_type; GLOBAL: global_variable_type;
                                                   s : natural; w : in out transition_stack_package.stack) is
begin
  null;
end Analyze_Predicates_Machine4;
                         .
                         .
                         .
separate (main)
procedure Analyze_Predicates_Machine8(local : machine8_state_type;. GLOBAL: global_variable_type;
                                                   s : natural; w : in out transition_stack_package.stack) is
begin
  null;
end Analyze_Predicates_Machine8;
```

Figure 22: Completed *Analyze_Predicates* procedures for the example ring protocol

```
separate(main)
procedure Action ( in_system_state : in out Gstate_record_type;
                   in_transition  : in out scm_transition_type;
                   out_system_state : in out Gstate_record_type )  is
begin
 case  in_transition  is  ——► Enabled transition
  when [redacted]  =>

    [redacted] ——► Action taken
•

•

•
  when others =>
    put("Error in the action procedure");
  end case;
end Action;
```

Figure 23: Template for *Action* procedure

```
separate (main)
procedure Action(in_system_state  : in out Gstate_record_type;  in_transition   : in out scm_transition_type;
                 out_system_state : in out Gstate_record_type) is
begin
   case (in_transition) is
      when (snd_data1) => out_system_state.GLOBAL_VARIABLES.CHAN1:=
                              in_system_state.machine1_state.out_buff1;
                          out_system_state.machine1_state.out_buff1 := E;

     when (rcv_data3) => out_system_state.machine1_state.in_buff1 :=
                             in_system_state.GLOBAL_VARIABLES.CHAN3;
                         out_system_state.machine1_state.out_buff1 := out_system_state.machine1_state.in_buff1;
                         out_system_state.GLOBAL_VARIABLES.CHAN3 :=E;

     when (snd_data2) =>   out_system_state.GLOBAL_VARIABLES.CHAN2:=
                              in_system_state.machine2_state.out_buff2;
                           out_system_state.machine2_state.out_buff2 := E;

     when (rcv_data1) =>  out_system_state.machine2_state.in_buff2 :=
                              in_system_state.GLOBAL_VARIABLES.CHAN1;
                          out_system_state.machine2_state.out_buff2 := out_system_state.machine2_state.in_buff2;
                          out_system_state.GLOBAL_VARIABLES.CHAN1 :=E;

     when (snd_data3) => out_system_state.GLOBAL_VARIABLES.CHAN3:=
                             in_system_state.machine3_state.out_buff3;
                         out_system_state.machine3_state.out_buff3 := E;

     when (rcv_data2) => out_system_state.machine3_state.in_buff3 :=
                             in_system_state.GLOBAL_VARIABLES.CHAN2;
                         out_system_state.machine3_state.out_buff3 := out_system_state.machine3_state.in_buff3;
                         out_system_state.GLOBAL_VARIABLES.CHAN2 :=E;

      when others => put_line("There is an error in the Action procedure");
   end case;
end Action;
```

Figure 24: Completed *Action* procedure for the example protocol

C. REACHABILITY ANALYSIS

The process of generating the set of all states reachable from the initial state is called reachability analysis. The program is capable of generating both the global and system reachability analyses separately for a protocol specified formally by the SCM model.

The user selects either *global reachability analysis* or *system state analysis* from a menu. During the graph construction, the program also detects the states with deadlock condition. Analysis results are stored in the output file named "rgraph.dat" in parallel with the graph construction.

Generating the global reachability analysis and system state analysis will be described in the following subsections.

1. Global Reachability Analysis

The structure of the global node representation used for the program is shown in Figure 25. This node structure also includes the outgoing transitions. The maximum number of outgoing transitions is limited to 7, which can be increased if necessary. The shared variables are stored in the *global_variables* variable and local variables are stored separately for each machine in the *machine_state** variables.

The initial global state is constructed from both the FSM text file and the initial values of the variables assigned in the *definitions* package. All the outgoing transitions are set to *null* initially. Starting with the initial global state, new nodes are added and linked to the graph. The algorithm for generating the global reachability graph is the same as the algorithm given for the system state analysis in Chapter II except that the "system states" must be replaced by "global states." Figure 26 shows a pseudo-code algorithm to construct the global reachability graph.

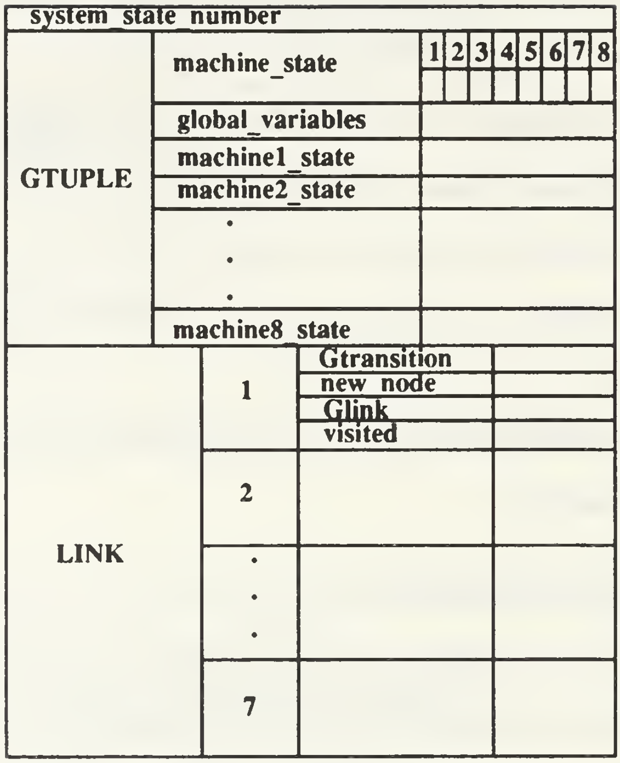

Figure 25: Global state structure with outgoing transitions

The program uses hashing for searching the reachability graph which increases the run time efficiency of the program. The reachability analysis is limited by the storage capacity of the computer and by the run time as in *Simple Mushroom* program. For example, the program generated 31,460 global states for a sliding window protocol of two machines defined in [Ref. 1] for a window size of 10. The run time for this example was about 10 minutes. The number of states and the run time increases greatly as the number of machines in the protocol increases and the protocol specifications become larger.

```
loop (main loop)
  for index1 in 1 .. total_number_of_machines loop
    position_holder(index1) := machine_array(index1) (M_state(index1))
    Determine the enabled transitions for the machine(index1) and push into transition_stack
    While not Empty(transition_stack) loop
      while (position_holder(index1) /= null) loop
        Traverse the machine arrays for each enabled transition in the stack
        if a transition found in the machine arrays create a temporary node resulting from this transition
          call Action procedure to make the necessary changes to the variables of this node
          Search the graph for this node
          if a node not found then
            insert and link the node to the graph
            Enqueue the node into the Gpointer_queue
          else
            link the node to the graph
          end if
        else
          position_holder(index1) := position_holder(index1).Slink
        end if
      end loop
      if not Empty(transition_stack) and a transition not found in the machine arrays
        pop the stack
      end if;
    end loop
  end loop
  if Gpointer_queue Empty then
    exit
  else
    Dequeue Gpointer_queue
    Update M_state for this new node
  end if
end loop (main loop)
```

Figure 26: Algorithm for generating global reachability graph for *Big Mushroom*

2. System State Analysis

The steps in constructing the system state graph are detailed in Chapter II. The structure of a system state is shown in Figure 27. Since the variables are not part of the system state, system state nodes are much smaller than the global state nodes. However, in order to determine the enabled transitions, variables are still needed for each node in the graph. The program stores the variables in secondary storage, instead of keeping them as a

part of the node, which decreases the amount of primary memory used and allows the analysis of larger and more complex protocols.

The pseudo-code algorithm for constructing the system reachability graph is shown in Figure 28.

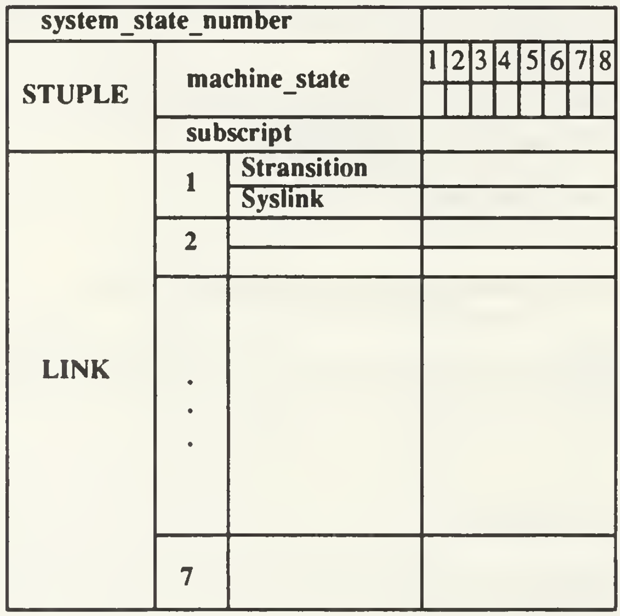

Figure 27: System state structure for *Smart Mushroom* program

D. OUTPUT

The program stores the results of the analysis in a file named “rgraph.dat.” This file contains FSMs in a tabular format, system/global reachability graph, and the results of the analysis consisting of number of states generated, number of states analyzed, and number of deadlocks. Unexecuted transitions are also listed at the end of the analysis.

Since each protocol specification has different variables, the user also has the flexibility to output the desired variables. This is done in a similar manner to the predicate-action table and variable definitions representation explained earlier using an Ada procedure template. The template for the *Output_Gtuple* procedure is shown in Figure 29.

The user completes the template with Ada "put" statements for outputting the global states. Since the system state tuples do not include the variables, there is no need to define an output format for system reachability graph.

```
loop (main loop)
  for index1 in 1.. num_of_trans loop
    if parent_Sstate.link(index1).Stransition /= unused then
      for index2 in 1 .. total_num_of_machines loop
        posiotion_holder := machine_array(index2) (M_state(index2))
        while position_holder /= null loop
          if position_holder.transition = parent_Sstate.link(index1).Stransition then
            create a temporary system state and store the corresponding variables
            determine the enabled outgoing transitions
            search the system state graph for this node
            if node not found then
              insert the node and link to the graph
              Enqueue the node into sys_pointer_queue
            else
              link the node to the graph
            end if
            exit
          else
            position_holder := position_holder.Slink
          end if
        end loop
        if an enabled transition found in the machine arrays then
          exit
        end if
      end loop
    else
      exit
    end if
  end loop
  if sys_pointer_queue empty then
    exit
  else
    Dequeue the sys_pointer_queue
    update M_state
  end if
end loop (main loop)
```

Figure 28: Algorithm for generating system state graph for *Smart Mushroom* program

The completed template for the *output_Gtuple* procedure is also given in Figure 30. As in *Simple Mushroom* program, if the analysis generates more than 2000 states, the program gives an interim summary and continues in steps as described in Chapter III. At the end of the program, the user can display/print the results or continue with another

system/global state analysis selecting the desired options from the menu. The output of the program for the example ring protocol is given in Figures 31 and 32.

```
separate (main)
procedure output_Gtuple (tuple : in out Gstate_record_type) is
begin
  if print_header then
    new_line(2);
    set_col(5);                 header format for the variables

    print_header := false;
  else
    put("[" & integer'image (tuple.machine_state (1)) );
    put(" , ");
                                machine 1 local variables

    put("[" & integer'image (tuple.machine_state (2)) );
    put(" , ");
      .

      .
    put("[" & integer'image (tuple.machine_state (8)) );
    put(" , ");
                                global variables

  end if;
end output_Gtuple;
```

Figure 29: Template for *output_Gtuple* procedure

```
separate (main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
  if print_header then
    new_line(2);
    set_col(5);
    put_line("  m1(in_buff1,out_buff1), m2(in_buff2,out_buff2),m3(in_buff3,out_buff3),
             (CHAN1,CHAN2,CHAN3)");
    print_header := false;
  else
    put("  [" & integer'image(tuple.machine_state(1)) );
    put(" , ");
    buff_enum_io.put(tuple.machine1_state.in_buff1);
    put(" , ");
    buff_enum_io.put(tuple.machine1_state.out_buff1);
    put(" ," & integer'image(tuple.machine_state(2)) );
    put(" , ");
    buff_enum_io.put(tuple.machine2_state.in_buff2);
    put(" , ");
    buff_enum_io.put(tuple.machine2_state.out_buff2);
    put(" , ");
    put(integer'image(tuple.machine_state(3)) );
    put(" , ");
    buff_enum_io.put(tuple.machine3_state.in_buff3);
    put(" , ");
    buff_enum_io.put(tuple.machine3_state.out_buff3);
    put(" , ");
    buff_enum_io.put(tuple.GLOBAL_VARIABLES.CHAN1);
    put(" , ");
    buff_enum_io.put(tuple.GLOBAL_VARIABLES.CHAN2);
    put(" , ");
    buff_enum_io.put(tuple.GLOBAL_VARIABLES.CHAN3);
    put(" ]");
  end if;

end output_Gtuple;
```

Figure 30: Completed *output_Gtuple* procedure for the example protocol

```
REACHABILITY ANALYSIS of :ring.scm
        SPECIFICATION


------------------------------
| Machine 1 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |   1  |  snd_data1   |
|   1   |   0  |  rcv_data3   |
------------------------------
------------------------------
| Machine 2 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |   1  |  rcv_data1   |
|   1   |   0  |  snd_data2   |
------------------------------
------------------------------
| Machine 3 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |   1  |  rcv_data2   |
|   1   |   0  |  snd_data3   |
------------------------------

GLOBAL REACHABILITY GRAPH

m1(in_buff1,out_buff1),m2(in_buff2,out_buff2),m3(in_buff3,out_buff3),(CHAN1,CHAN2,CHAN3)

 0   [ 0 , E , D , 0 , E , E ,  0 , E , E , E , E , E ] snd_data1   1
 1   [ 1 , E , E , 0 , E , E ,  0 , E , E , D , E , E ] rcv_data1   2
 2   [ 1 , E , E , 1 , D , D ,  0 , E , E , E , E , E ] snd_data2   3
 3   [ 1 , E , E , 0 , D , E ,  0 , E , E , E , D , E ] rcv_data2   4
 4   [ 1 , E , E , 0 , D , E ,  1 , D , D , E , E , E ] snd_data3   5
 5   [ 1 , E , E , 0 , D , E ,  0 , D , E , E , E , D ] rcv_data3   6
 6   [ 0 , D , D , 0 , D , E ,  0 , D , E , E , E , E ] snd_data1   7
 7   [ 1 , D , E , 0 , D , E ,  0 , D , E , D , E , E ] rcv_data1   8
 8   [ 1 , D , E , 1 , D , D ,  0 , D , E , E , E , E ] snd_data2   9
 9   [ 1 , D , E , 0 , D , E ,  0 , D , E , E , D , E ] rcv_data2  10
10   [ 1 , D , E , 0 , D , E ,  1 , D , D , E , E , E ] snd_data3  11
11   [ 1 , D , E , 0 , D , E ,  0 , D , E , E , E , D ] rcv_data3   6

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------------
Number of states generated :12
Number of states analyzed :12
Number of deadlocks : 0
                                   UNEXECUTED TRANSITIONS
                                       *****NONE*****
```

Figure 31: Program output for global reachability analysis

```
REACHABILITY ANALYSIS of :ring.scm

SPECIFICATION
------------------------------
| Machine 1 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |   1  |  snd_data1   |
|   1   |   0  |  rcv_data3   |
------------------------------
------------------------------
| Machine 2 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |   1  |  rcv_data1   |
|   1   |   0  |  snd_data2   |
------------------------------

------------------------------
| Machine 3 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |   1  |  rcv_data2   |
|   1   |   0  |  snd_data3   |
------------------------------

SYSTEM REACHABILITY GRAPH
0 [ 0, 0, 0 ] 0   snd_data1   1
1 [ 1, 0, 0 ] 0   rcv_data1   2
2 [ 1, 1, 0 ] 0   snd_data2   3
3 [ 1, 0, 0 ] 1   rcv_data2   4
4 [ 1, 0, 1 ] 0   snd_data3   5
5 [ 1, 0, 0 ] 2   rcv_data3   0

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------------
Number of states generated :6
Number of states analyzed :6
Number of deadlocks : 0


UNEXECUTED TRANSITIONS
*****NONE*****
```

Figure 32: Program output for system state analysis[2]

2. The number next to "]" sign shows the subscripts that is explained in Chapter II.

V. EXAMPLES FOR USING THE MUSHROOM PROGRAM

In this Chapter, the programs *Simple Mushroom*, *Big Mushroom*, and *Smart Mushroom* are demonstrated with several examples.

The *Simple Mushroom* program will be used to analyze a simple example four machine protocol which illustrates some important aspects of the program, such as detecting unspecified receptions, unexecuted transitions etc. Also, the information transfer phase of a full duplex LAP-B protocol specified by the CFSM model will be analyzed. This protocol illustrates a larger and more complex analysis.

The *Big Mushroom* and *Smart Mushroom* programs will be used to analyze the *GO BACK N* protocol with a window size of 10, and the *Token Bus* protocol, which illustrates some important aspects of the *system state analysis*.

A. CFSM MODEL

1. A Simple Four Machine Protocol

The specification of the protocol using the CFSM model is shown in Figure 33. Each of the machines sends/receives a message/acknowledgment from another machine. Machines 2 and 3 also have another send transition from state 1 to state 3. The FSM description of the protocol is shown in Figure 34, and analysis results obtained by the *Simple Mushroom* program are shown in Figure 35. The analysis generated 36 global states. There are three unspecified receptions and one unexecuted transition. No deadlocks or channel overflows are recorded. The maximum channel size is 2. These results are obtained by simply entering the FSM text file into the program. This analysis would be very cumbersome to do manually, even for a simple specification like this one.

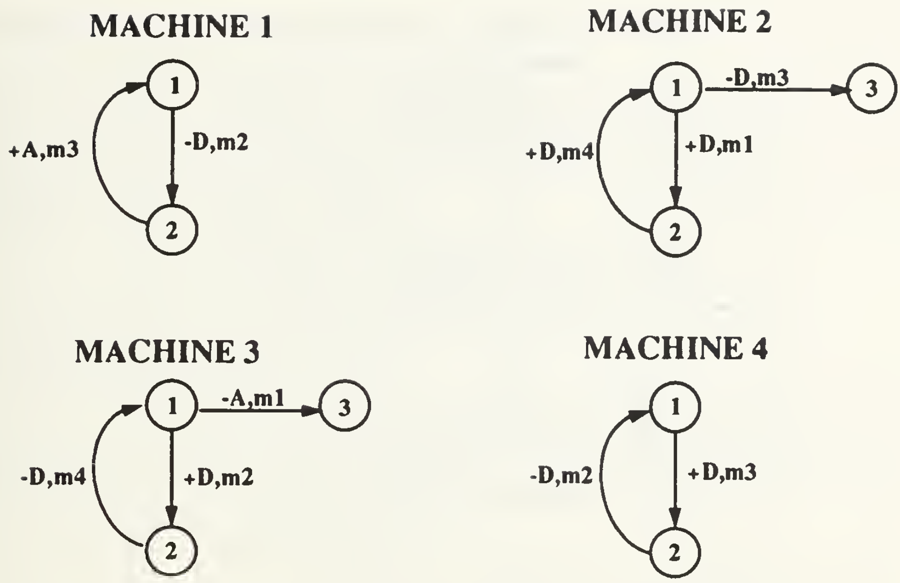

Figure 33: Specification for the example four machine protocol

```
start
number_of_machines 4
machine 1
state 1
trans -D 2 2
state 2
trans +A 1 3
machine 2
state 1
trans -D 3 3
trans +D 2 1
state 2
trans +D 1 4
machine 3
state 1
trans -A 3 1
trans +D 2 2
state 2
trans -D 1 4
machine 4
state 1
trans +D 2 3
state 2
trans -D 1 2
initial_state 1 1 1 1
finish
```

Figure 34: FSM text file for the example protocol

REACHABILITY ANALYSIS of : four_machine.fsm

SPECIFICATION

Machine 1 State Transitions			
From	**To**	**other machine**	**Transition**
1	2	2	s D
2	1	3	r A

Machine 2 State Transitions			
From	**To**	**other machine**	**Transition**
1	3	3	s D
1	2	1	r D
2	1	4	r D

Machine 3 State Transitions			
From	**To**	**other machine**	**Transition**
1	3	1	s A
1	2	2	r D
2	1	4	s D

Machine 4 State Transitions			
From	**To**	**other machine**	**Transition**
1	2	3	r D
2	1	2	s D

REACHABILITY GRAPH

```
 1 [ 1,E,E,E, 1,E,E,E, 1,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 1,E,E,E, 1,E,E,E, 1,E,E,E]         2
     -D   3  [ 1,E,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]         3
     -A   1  [ 1,E,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]         4
 2 [ 2,D  ,E,E, 1,E,E,E, 1,E,E,E, 1,E,E,E]
     -D   3  [ 2,D  ,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]       5
     +D   1  [ 2,E,E,E, 2,E,E,E, 1,E,E,E, 1,E,E,E]           6
     -A   1  [ 2,D  ,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]       7
 3 [ 1,E,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]       5
     -A   1  [ 1,E,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]       8
     +D   2  [ 1,E,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]           9
 4 [ 1,E,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]       7
     -D   3  [ 1,E,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]       8
 5 [ 2,D  ,E,E, 3,E,D  ,E, 1,E,E,E, 1,E,E,E]
     -A   1  [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]     10
     +D   2  [ 2,D  ,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]         11
 6 [ 2,E,E,E, 2,E,E,E, 1,E,E,E, 1,E,E,E]
     -A   1  [ 2,E,E,E, 2,E,E,E, 3,A  ,E,E, 1,E,E,E]         12
 7 [ 2,D  ,E,E, 1,E,E,E, 3,A  ,E,E, 1,E,E,E]
     +A   3  [ 1,D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]         13
     -D   3  [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]     10
     +D   1  [ 2,E,E,E, 2,E,E,E, 3,A  ,E,E, 1,E,E,E]         12
 8 [ 1,E,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]     10
 9 [ 1,E,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  ,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]         11
     -D   4  [ 1,E,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]         14
10 [ 2,D  ,E,E, 3,E,D  ,E, 3,A  ,E,E, 1,E,E,E]
     +A   3  [ 1,D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]       15
11 [ 2,D  ,E,E, 3,E,E,E, 2,E,E,E, 1,E,E,E]
     -D   4  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]       16
12 [ 2,E,E,E, 2,E,E,E, 3,A  ,E,E, 1,E,E,E]
     +A   3  [ 1,E,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]           17
13 [ 1,D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]
     -D   2  [ 2,D  D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]      18
     -D   3  [ 1,D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]       15
     +D   1  [ 1,E,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]           17
```

```
14 [ 1,E,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]       16
      -A   1  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]       19
      +D   3  [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]           20
15 [ 1,D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]    21
16 [ 2,D  ,E,E, 3,E,E,E, 1,E,E,D  , 1,E,E,E]
      -A   1  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]     22
      +D   3  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]         23
17 [ 1,E,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]
      -D   2  [ 2,D  ,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]         24
18 [ 2,D  D  ,E,E, 1,E,E,E, 3,E,E,E, 1,E,E,E]
      -D   3  [ 2,D  D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]    21
      +D   1  [ 2,D  ,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]         24
19 [ 1,E,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]     22
      +D   3  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]         25
20 [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]         23
      -A   1  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]         25
      -D   2  [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]         26
21 [ 2,D  D  ,E,E, 3,E,D  ,E, 3,E,E,E, 1,E,E,E]**********Unspecified Reception***********
22 [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,D  , 1,E,E,E]
      +A   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]       27
      +D   3  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]       28
23 [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 2,E,E,E]
      -A   1  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]       28
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]       29
24 [ 2,D  ,E,E, 2,E,E,E, 3,E,E,E, 1,E,E,E]**********Unspecified Reception***********
25 [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]       28
      -D   2  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]       30
26 [ 1,E,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]       29
      -A   1  [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]       30
27 [ 1,D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]    31
      +D   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]         32
28 [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 2,E,E,E]
      +A   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]         32
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]     33
29 [ 2,D  ,E,E, 3,E,E,E, 1,E,E,E, 1,E,D  ,E]
      -A   1  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]     33
30 [ 1,E,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]
      -D   2  [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]     33
31 [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,D  , 1,E,E,E]
      +D   3  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]      34
32 [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]      34
      -D   2  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]       35
33 [ 2,D  ,E,E, 3,E,E,E, 3,A  ,E,E, 1,E,D  ,E]
      +A   3  [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]       35
34 [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 2,E,E,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]    36
35 [ 1,D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]
      -D   2  [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]    36
36 [ 2,D  D  ,E,E, 3,E,E,E, 3,E,E,E, 1,E,D  ,E]**********Unspecified Reception***********

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
-----------------------------------------------------
Total number of states generated : 36
Number of states analyzed : 36
number of deadlocks : 0
number of unspecified receptions : 3
maximum message queue size :  2
channel overflow :NONE
                              UNEXECUTED TRANSITIONS
               ----------------------------------------------------
               |         Machine  2  Unexecuted Transitions       |
               ----------------------------------------------------
               | From  |  To  | other machine | Unexecuted Transition |
               ----------------------------------------------------
               |  2    | 1    |      4        |        r D        |
               ----------------------------------------------------
```

Figure 35: Program output for the example protocol

2. Analysis of Information Transfer Phase of the LAP-B Protocol

In this Section, analysis of a Data Link Control (DLC) protocol is described using the *Simple Mushroom* program. The LAP-B protocol is modeled and analyzed with CFSM model [Ref. 14]. A simplified analysis of the information transfer phase of the protocol, which includes only I-frames with a window size of 2, will be described below.

This analysis is important in two ways. First, it verifies that the program is correct by obtaining the same analysis results as in [Ref. 14]. Secondly, it is a good example to show that the total number of global states can be very large, even for such a limited protocol. The description of the information transfer phase is explained below as it appears in [Ref. 14].

The network nodes, which are connected by the protocol, consist of a Data Terminal Equipment (DTE) and a Data Circuit Terminating Equipment (DCE). In this model, DTE and DCE are considered process 1 and process 2 respectively. Each of these processes are also modeled as three sub-processes: *Sender*, *Receiver* and *Frame Assembler Disassembler* (FAD), which are numbered as 1 or 2 according to their process numbers.

Figure 36 shows the processes and how they are connected. The FAD process combines data blocks from the Sender with acknowledgments from the Receiver, into complete I-frames and sends the I-frames to the FAD of the other process. The FAD also breaks up the I-frames received from the other FAD and sends the acknowledgment to the Sender, and data blocks to the Receiver.

I-frames are expressed by the notation “Inm”, where n is the send sequence number N(S), and m is the receive sequence number N(R). The message “Di” is a data block sent from the Sender to the FAD, or from the FAD to the receiver; it is the data block which is to be placed in, or which is taken out of, the I-frame. The “i” in “Di” is the send sequence number. The message “Ai” is an acknowledgment with a receive sequence number of i.

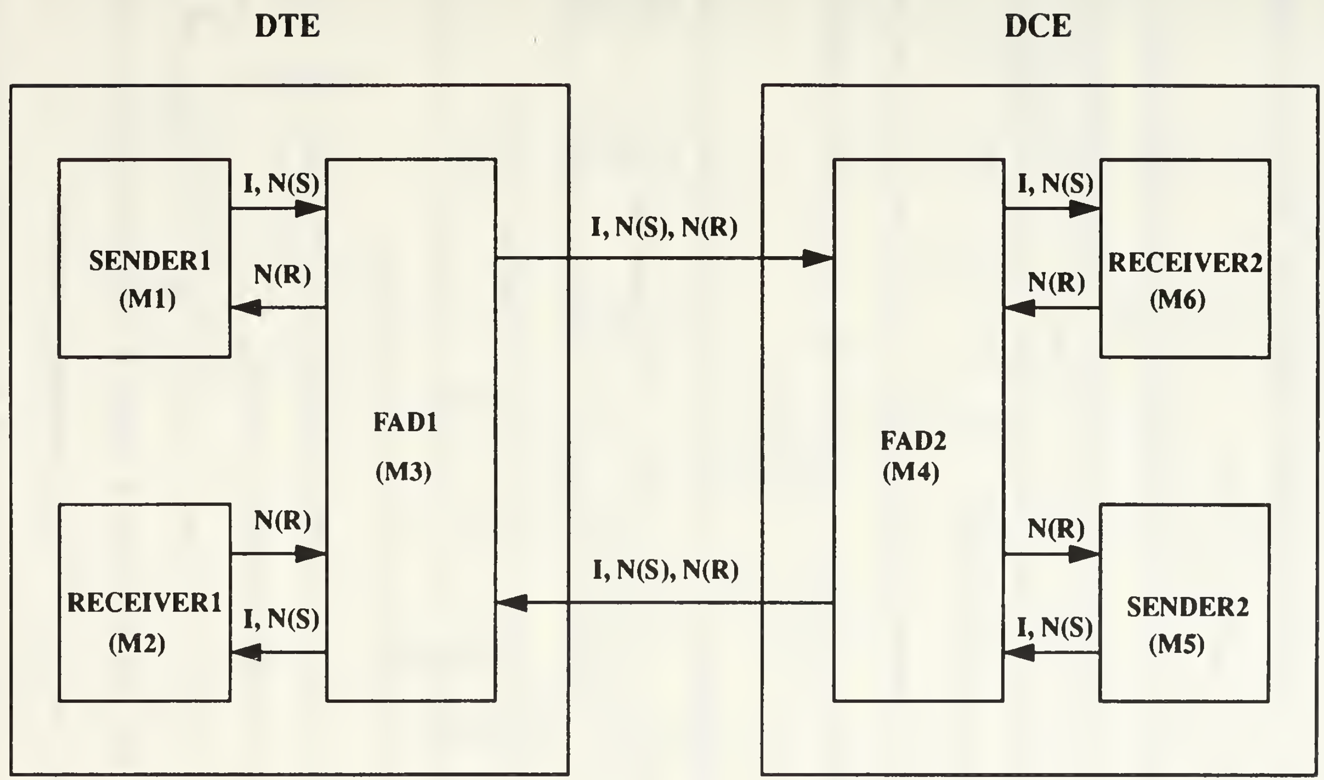

Figure 36: Processes for the Information Transfer Phase

The finite state machines for the Sender, Receiver and FAD of the DTE are shown in Figures 37, 38 and 39. The FSMs for the DCE are the same except that FAD1, RECEIVER1, and SENDER1 must be replaced with FAD2, RECEIVER2, and SENDER2 respectively. Since no RR-frames are used, I-frames can only be acknowledged by receiving an N(R) from an incoming I-frame.

As an example, suppose the DTE Sender1 has 3 data blocks to send. It can go from state 1 to state 2, sending "D0," and then to state 3, sending the second block as "D1." At this point, 2 data blocks are outstanding, so it must wait for an acknowledgment of at least one of them before sending the third.

The DTE FAD1 process, initially in state 1, will receive the D0 from Sender1 and enter state 2. It then sends an "enquiry" to the Receiver1 to get the latest acknowledgment, an N(R), for the data blocks received from the DCE.

Since no data blocks have been received by the DTE yet, Receiver1 will respond with an "A0." FAD1 will receive the A0, and will transition from state 8 to 11. The FAD1 will then return to state 1 sending the I-frame "I00." Similarly, the FAD1 will receive the second data block, D1, and transmit it as "I10" after combining with "A0."

FAD2 will receive the "I00" frame first, entering state 20. It then splits this I-frame and sends the "D0" to Receiver2, and "A0" to Sender2.

Sender2 is in state 1, and simply discards this "A0." Receiver2 is in state 1, accepts the "D0" data block and transitions to state 2.

Similarly, The DCE FAD2 process receives the "I10" message, and sends the "D1" to Receiver 2, and "A0" to Sender 2. Sender 2 will discard the "A0", remaining in state 1, and Receiver 2 will receive "D1," transitioning to state 3.

Suppose at this point a user data block becomes available to send at the DCE. It will send an "I02" frame across the data link to the DTE; and upon receiving the I02, the DTE will now be able to send the third user data block.

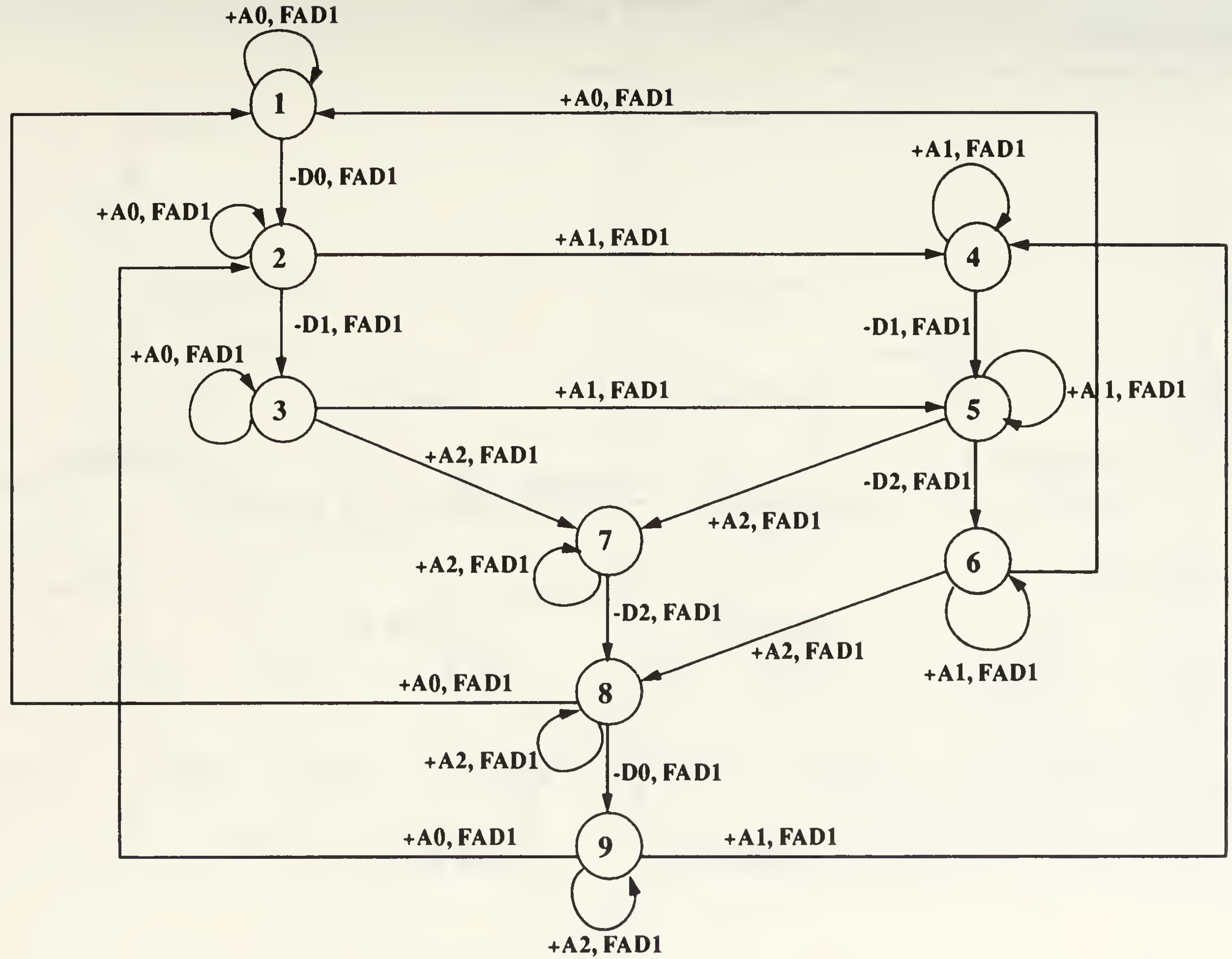

Figure 37: Sender 1 [Ref. 14]

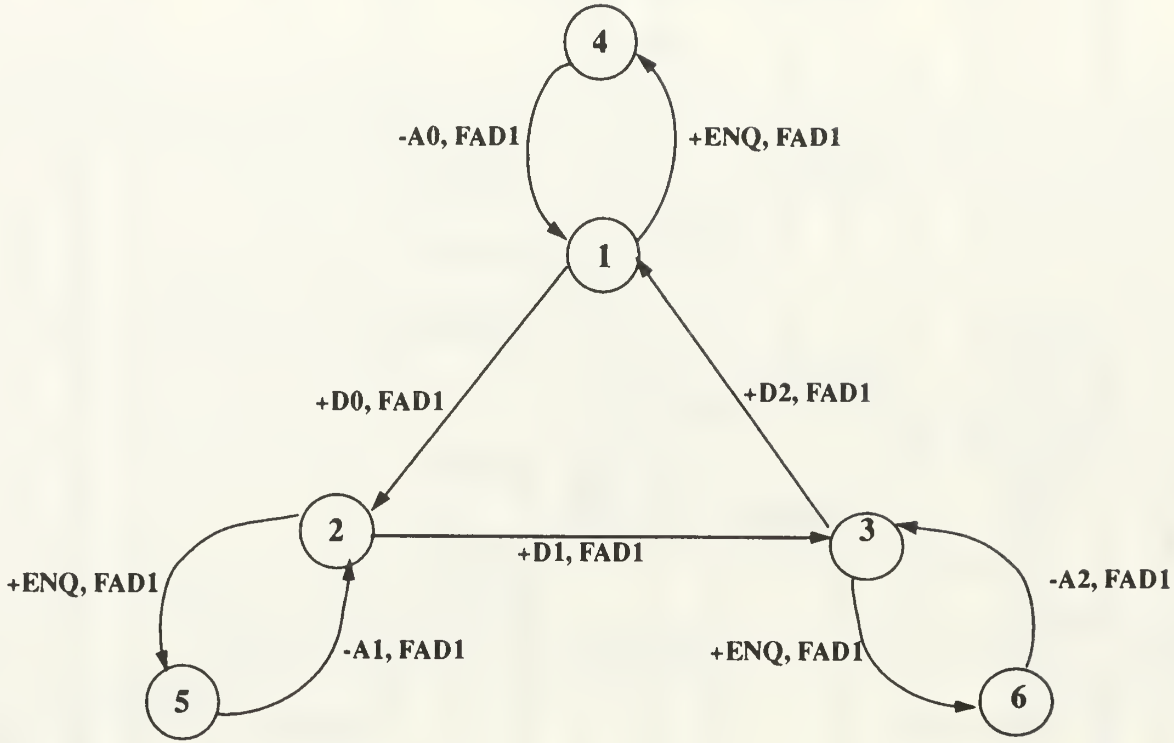

Figure 38: Receiver 1 [Ref. 14]

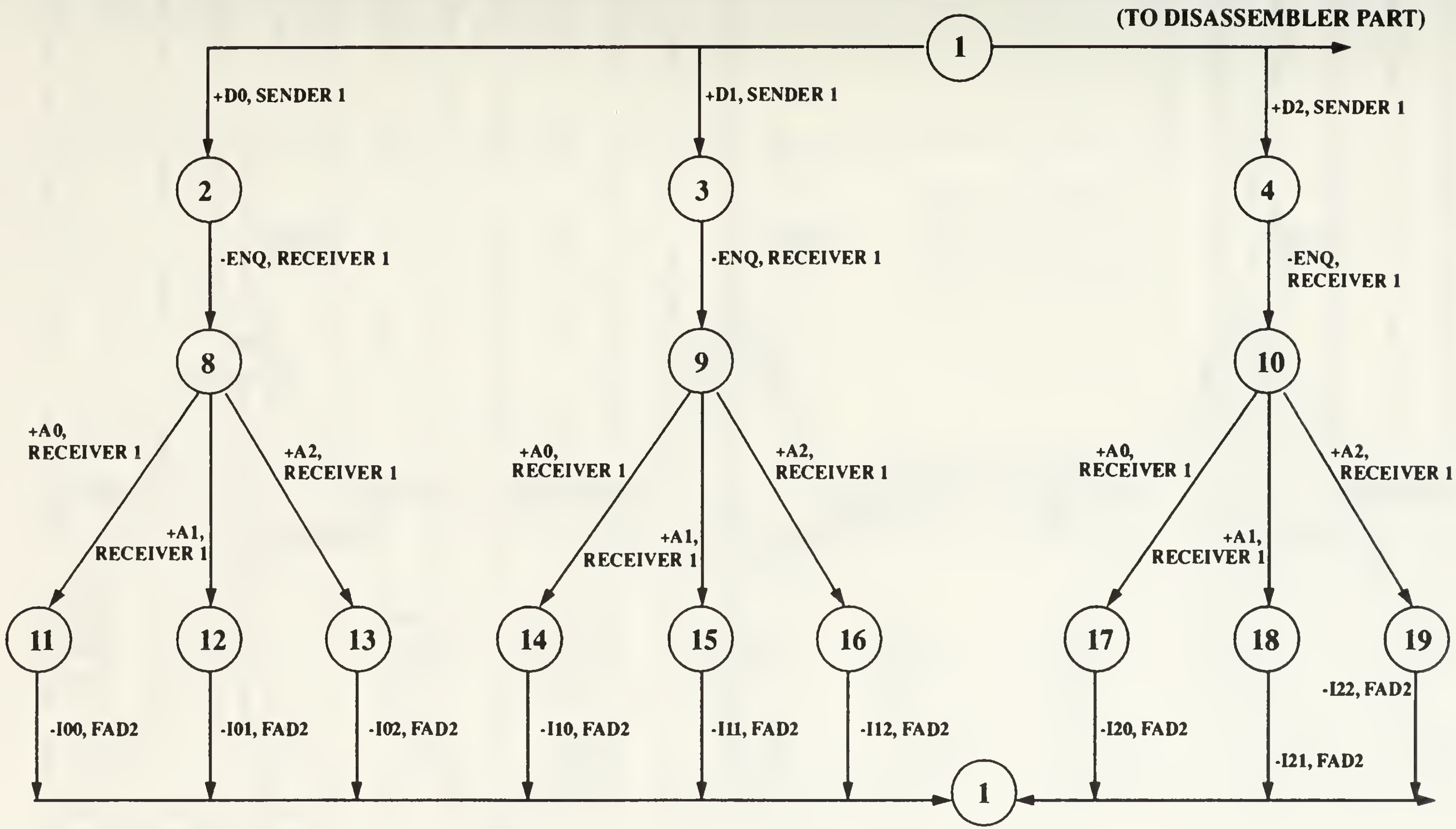

Figure 39a: Frame Assembler Disassembler FAD1 (Assembler Part) [Ref. 14]

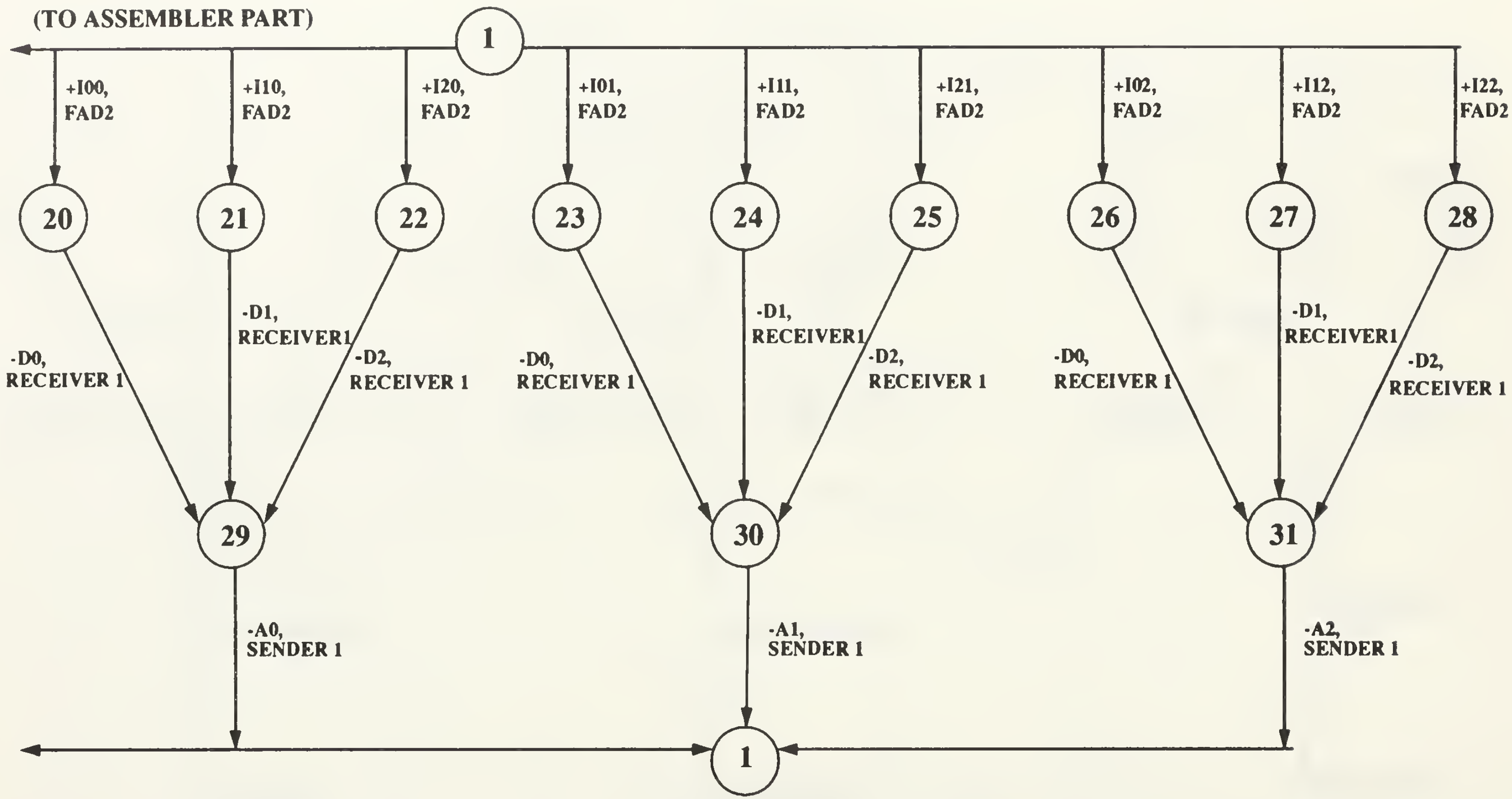

Figure 39b: Frame Assembler Disassembler FAD1 (Disassembler Part) [Ref. 14]

For the automated analysis of the protocol, the FSMs in Figures 37, 38, and 39 are converted to a text file and entered into the program as shown in Appendix A. The transition names in this text file are the same as in the FSM diagrams, such as "+I00", "+D0" etc. In order to save memory and generate a larger number of states in the analysis, the transition names can be abbreviated to single characters at the time of the analysis as shown below:

D0 -> X	I00 -> 1
D1 -> Y	I01 -> 2
D2 -> Z	I02 -> 3
A0 -> A	I10 -> 4
A1 -> B	I11 -> 5
A2 -> C	I12 -> 6
ENQ -> Q	I20 -> 7
	I21 -> 8
	I22 -> 9

The amount of memory available and the CPU time are always a concern for a full reachability analysis. The program output for the analysis is partially given in Appendix A. Because of the size of the analysis, only a very small portion of the reachable states are included in the output. The total number of global states generated for the information phase was 73391. There were no unspecified receptions, unexecuted transitions, and channel overflows. The maximum channel length was 6. A deadlock condition was found at state 17034 where all the channels were empty and Sender1, Receiver1, FAD1, FAD2, Sender2, Receiver2 were in states 3, 3, 1, 1, 3, 3 respectively. This state deadlock is expected since RR-frames are not included in the analysis. A more detailed explanation including the RR-frames in the protocol is given in [Ref. 14]. The reader may note that the results of the analysis exactly match with the results reported in Reference 14. The deadlock state found in Reference 14 was 67699, which was recorded at state 17034 in this analysis. However, the global states are the same for both analyses. The *Simple Mushroom* program uses a *Breadth-First Search* algorithm for choosing the states from the work set

(i.e, global states that are generated, but have not been analyzed yet). The protocol verifier PROVE, used in Reference 14 might be using a *Depth First Search* approach, which would result in a different global state number.

The protocol, including the RR-frames, was also entered into the program, but the program could not complete the analysis due to insufficient computer memory. In this analysis, 153565 global states were generated. No unspecified receptions, deadlocks or channel overflows were recorded for the analyzed portion of the protocol. The maximum channel size reached was 4. The program completed the analysis in 11 hours 51 minutes on a Sun SPARC station.

B. SCM MODEL

1. Go Back N

The first protocol selected for analysis using the *Big Mushroom* and *Smart Mushroom* programs is a 1-way data transfer protocol with a variable window size, which is essentially a subset of the High-level Data Link Control (HDLC) class of protocols. This protocol is modeled and analyzed with the SCM model in [Ref. 1]. The same specification will be used here and an automated analysis will be described using the programs developed for a window size of 10. The specification is summarized below:

There are two machines in the system, a sender (m_1) and a receiver (m_2). The sender sends data blocks to the receiver, which are numbered sequentially, 0, 1,..., w, 0, 1, ... for a window size of *w*. As in HDLC, the maximum number of data blocks which can be sent without receiving an acknowledgment is *w*, the window size. The receiver, m_2, receives the data blocks and acknowledges them by sending the sequence number of the next data block expected (which is stored in local variable *exp*). The shared variables DATA and SEQ are used to pass messages from sender to receiver, and the shared variable

ACK is used to pass acknowledgments back to the sender. The receiver may acknowledge any number of blocks received up to the window size. Upon receiving the acknowledgment, the sender must be able to deduce how many data blocks are being acknowledged. This is done by observing the difference between the values of the received acknowledgment and the sequence number of the last data block sent.

The general specification of the protocol is given in Figure 40 and in Table 4. Initially, both sender and receiver are in state 0, arrays DATA and SEQ are empty, and ACK is empty. The domains of DATA, *Rdata* and *Sdata* are not specified; these are used to hold user data blocks. *Sdata* and *Rdata* are the interface or access points of the higher layer (user) protocol. The local variables for the sender are *Sdata*, used to store data blocks, *seq*, used to store the sequence number of the next data block to be sent out, and *i*, used as an index into the DATA and SEQ arrays. Initially *seq* is set to 0, and *i* is set to 1. The local variables of the receiver are *Rdata*, *exp*, and *j*. *Rdata* is used to receive and store incoming data blocks, *exp* to hold the expected sequence number of the next incoming data block, and *j* is an index into the shared arrays DATA and SEQ.

The states of both sender and receiver are numbered 0, 1, ..., *w*, and each state has an easily recognized intuitive meaning. If the sender is in state 0, then all data blocks sent to date have been received by the receiver, so a full window size of *w* data blocks may be sent without waiting for an acknowledgment. If m_1 is in state *w*, then a full window of blocks have been sent, so the sender can only wait for the acknowledgment from the receiver.

If the receiver, m_2, is in state 0, then all received data blocks have been acknowledged. If in state *w*, then a full window of data blocks have been received, but not acknowledged. Whenever the receiver sends an acknowledgment, all data blocks received up to that point are acknowledged.

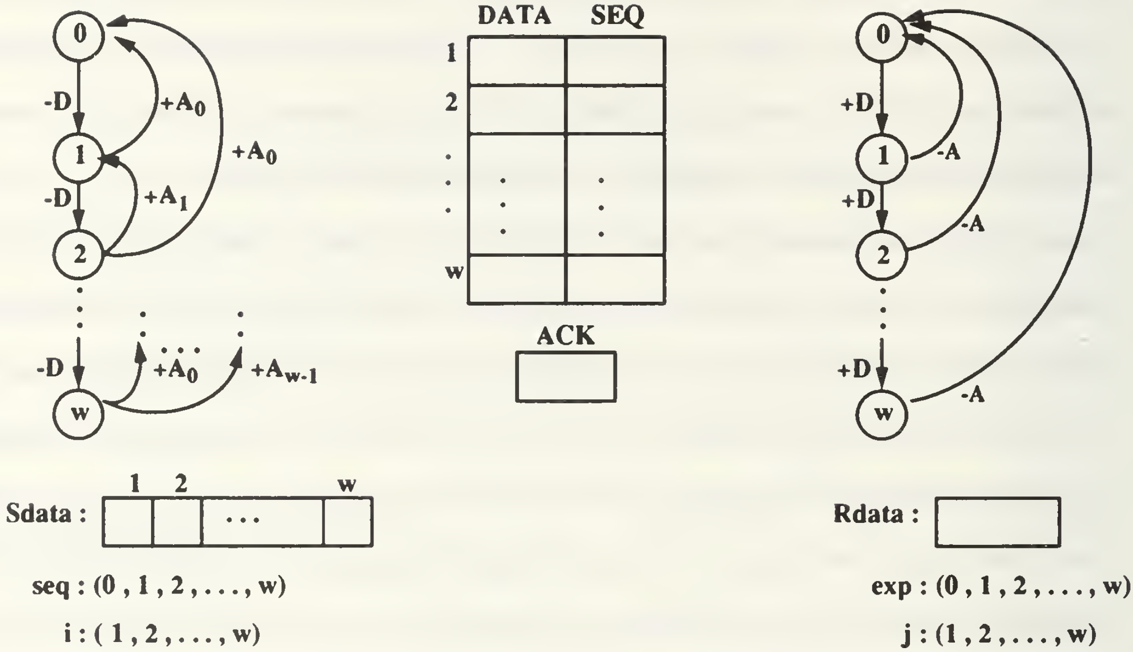

Figure 40: State machines and variables for *Go Back N*

TABLE 4: PREDICATE-ACTION TABLE FOR *GO BACK N*

Transition	Enabling Predicate	Action
-D	DATA(i) = ε ∧ SEQ(i) = ε	DATA(i) ← *Sdata*(i) SEQ(i) ← *seq* *inc*(i, *seq*)
+A_k ($0 \le k \le w$)	ACK ⊕ k = *seq* ∧ ACK ≠ ε (next state : k)	ACK ← ε
+D	DATA(j) ≠ ε ∧ SEQ(j) = *exp*	*Rdata* ← DATA(j) DATA(j), SEQ(j) ← ε *inc* (j, *exp*)
-A	DATA(j) = ε	ACK ← *exp* *Rdata* ← ε

The enabling predicate and action for each transition are shown in Table 4. The label or transition name is the leftmost column, the enabling predicate in the middle, and the corresponding action on the right. There are four basic types of transitions. In the sender, m_1, the $-D$ transition transmits a data block by placing it into the shared variable DATA(i), and the sequence number into SEQ(i). The send is enabled whenever those variables are empty. (The interaction between the sender and the user, or higher layer, is implicit, and not specified here). The *inc* operation increments its arguments, if less than their maximum value, in which case it resets them to the minimum value. The operator $\oplus$ represents the *inc* operation repeated k times, if the argument is k and the symbol ε denotes the empty value. The receive transition in the receiver, m_2, is enabled whenever a data block of the appropriate sequence number is in the jth element of DATA and SEQ. An acknowledgment may be sent by m_2 in any state except 0, in which case no unacknowledged data blocks have been received.

The remaining transition is the $+A_k$ receive acknowledgment, in m_1. If m_1 is in state u, $1 \leq u \leq w$, and there is a nonempty value in shared variable ACK, then exactly one of the transitions $+A_0$, $+A_1$, ..., $+A_{w-1}$ will be enabled; it will be that A_k such that the predicate ACK$\oplus k = seq$ is true, and the next state is k. [Ref. 1]

For analyzing this protocol using the *Big Mushroom* and *Smart Mushroom* programs, the inputs to the program must be completed. These consist of a text file description of FSMs, the package, *definitions*, which include the variables of the protocol, and the subprograms *Analyze_Predicates_Machines* and *Action*, which define the predicate-action table. Also an *Output_Gtuple* procedure, which defines the output format for the global tuples, must be entered. Completed packages/procedures for a window size of 10 are given in Appendix B.

The same names are used for local and shared variables in the package *definitions* as in the predicate-action table. Variables DATA, ACK and *Sdata* are declared as one

dimensional arrays of size 10, which is the window size. Local variables *seq* and *exp* and index numbers *i* and *j* are declared as integers in the range 0 to 10. Global variable ACK is declared as integer in the range -1 to 10, where -1 represents ε value in the predicate-action table. An enumeration type, *buffer_type,* is declared for storing the data passed by the upper layer to local variable *Sdata*. Data are declared as d0, d1, .., d9,*e*, where *e* represents the ε value. Transition names in the specification are defined as *snd_data*, *rcv_data*, *snd_ack*, *rcv_acki* for *-D*, *+D*, *-A*, and $+A_i$ in predicate-action table respectively.

Actions and *predicates* are also translated to Ada statements in the subprograms *Analyze_predicates_Machines* and *Action*. For each state in both machines there is a "when" statement. The predicates for the outgoing transitions from that state are translated to Ada with "if" conditional statements. Actions in the predicate-action table are converted to Ada statements with "when" statements (see Appendix B).

The program generated 286 system states and 31,460 global states, which are identical with the results obtained by the formulas given in [Ref. 1]. The protocol is free from deadlocks and there are no unexecuted transitions. The difference between the number of system and global states shows the power of the system state analysis which reduced the number of states in the reachability graph exponentially. However, without the *Smart Mushroom* program, the system state analysis would be cumbersome to do manually, and the global reachability analysis would be infeasible.

2. Token Bus

Another example of the program application, the token bus specification in [Ref. 15] will be used. The specification is a simplified one. It assumes that the transmission medium is error free and all transmitted messages are received undamaged. Both the system state analysis and global analysis are generated from this token bus specification for a protocol consisting of 8 machines.

The specification of this simplified protocol is given in Figure 41 and Table 5. The FSM diagram and the local variables are the same for each machine, where the transition names: *ready*, *rcv*, *pass*, *get-tk*, *pass-tk*, *Xmit*, and *moreD* are appended with the corresponding machine number to the end for each machine in the specification. For example, transitions for machine 7 are named as *ready7*, *rcv7*, *pass7*, etc. This makes it easier to follow the reachability graphs. The remainder of the protocol specification as described in Reference 15 is as follows: The shared variable, *MEDIUM*, is used to model the bus, which is "shared" by each machine. A transmission onto the bus is modeled by a write into the shared variable. The fields of this variable correspond to the parts of the transmitted message: the first field, *MEDIUM.T*, takes the values T or D, which indicate whether the frame is a token or a data frame. The second field contains the address of the station to which the message is transmitted (DA for "destination address"); the next field, the originator (SA for "source address"); and finally the data block itself.

The network stations, or machines, are defined by a finite state machine, a set of local variables, and a predicate-action table. The *initial state* of each machine is state 0, and the shared variable is initially set to contain the token with the address of one of the stations in the "DA" field.

The value of local variable *next* is the address of the next or downstream neighbor, and these are initialized so that the entire network forms a cycle, or logical ring.

The local variable *i* is used to store the station's own address. As implied by the names, the local variables *inbuf* and *outbuf* are used for storing data blocks to be transmitted to or retrieved from other machines on the network. The latter of these, *outbuf*, is an array and thus can store a potentially large number of data blocks. The local variable *ctr* serves to count the number of blocks sent; it is an upper bound on the number of blocks which can be sent during a single token holding period. The local variable *j* is an index into the array *outbuf*.

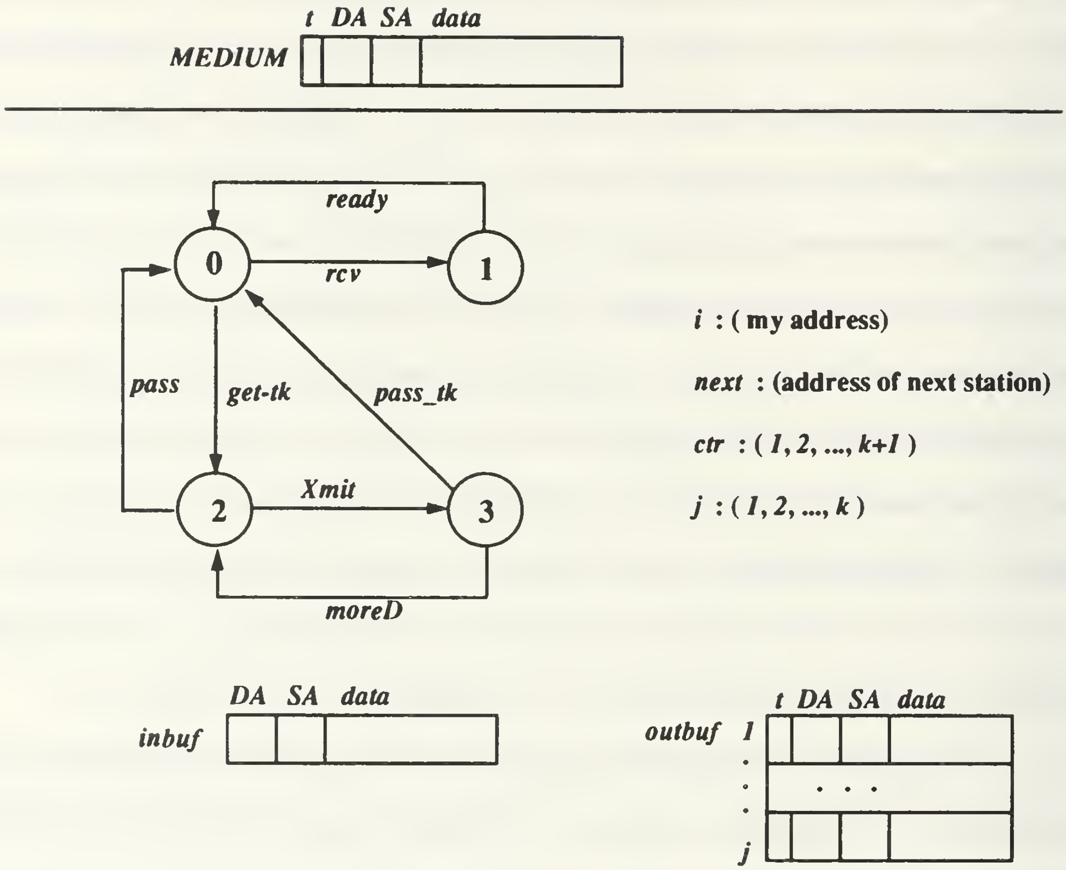

Figure 41: FSM and variables for the network nodes

The local variables *j* and *ctr* are initially set to 1, and *inbuf* and *outbuf* are initially set to empty. The shared variable *MEDIUM* initially contains the token, with the address of the station in the DA field. Thus the initial system state tuple is (0,0, ..., 0) and the first transition taken will be *get-tk* by the station which has its local variable *i* equal to *MEDIUM.DA*.

Each machine has four states. In the initial state, 0, the stations are waiting to either receive a message from another station, or the token. If the token appears in the variable *MEDIUM* with the station's own address, the transition to state 2 is taken. When

taking the *get-tk* transition, the machine clears the communication medium and sets the message counter *ctr* to 1. In state 2, the station transmits any data blocks it has, moving to state 3, or passes the token, returning to state 0. In state 3, the station will return to state 2 if any additional blocks are to be sent, until the maximum count k is reached. When the count is reached, or when all the station's messages have been sent, the station returns to state 0.

The receiving station, as with all stations not in possession of the token, will be in state 0. The message will appear in *MEDIUM*, with the receiving station's address in the *DA* field. The receiving transition to state 1 will then be taken, the data block copied, and *MEDIUM* cleared. By clearing the medium, the receiving station enables the sending station to return to its initial state (0) or to its sending state (2).

TABLE 5: PREDICATE-ACTION TABLE FOR THE NETWORK NODES

Transition	Enabling Predicate	Action
rcv	$MEDIUM.(t, DA) = (D, i)$	$inbuf \leftarrow MEDIUM.(SA, data)$
ready	*true*	$MEDIUM \leftarrow \phi$
get-tk	$MEDIUM.(t, DA) = (T, i)$	$MEDIUM \leftarrow \phi$; $ctr \leftarrow 1$
pass	$outbuf[j] = \phi$	$MEDIUM \leftarrow (T, next, i, \phi)$
Xmit	$outbuf[j] \neq \phi$	$MEDIUM \leftarrow outbuf[j]$; $ctr \leftarrow ctr \oplus 1$; $j \leftarrow j \oplus 1$ $outbuf[j] \leftarrow \phi$
moreD	$MEDIUM = \phi \wedge outbuf[j] \neq \phi$	*null*
pass-tk	$MEDIUM = \phi \wedge$ $(outbuf[j] = \phi \vee ctr = k + 1)$	$MEDIUM \leftarrow (T, next, i, \phi)$

The symbol "⊕" indicates that the variable should be incremented unless its maximum value has been reached, in which case it should be reset to the initial value. The notation *MEDIUM*.(*t*, *DA*) is used to denote the first two fields of the variable *MEDIUM*. For example, *MEDIUM*.(*t*, *DA*) = (*T*, *i*) is a boolean expression which is true if and only if the first field of *MEDIUM* contains the value *T*, and the second field contains the value *i*. Other notations in the predicate-action table such as "∧", "∨", "←" etc. are intuitive.

The inputs to the program for the reachability analysis of this protocol are given in Appendix C. The same names as in the specification are used for the local and global variables in the package *definitions*. Also, the "empty" value is represented by "E" and the data are represented by "I" in this package. The upper bound on the number of data blocks in the *outbuf* variable is set to 7.

The system state analysis alone did not give a complete analysis due to some loops in the FSMs of the SCM specification. Since the system state analysis assumes that two system states are equivalent if both the machine state tuples and the outgoing transitions are the same, this can cause the system state analysis to give insufficient results in some special cases. For example, incomplete results can arise when the FSMs of the specification include some loops that result with the same states and enabled transitions repeatedly. In such specifications, some of the transitions will stay unexecuted, resulting an incomplete analysis. This situation is observed in this specification when one of the machines had two or more data blocks in its *outbuf* local variable. For instance, if machine 1 has two data blocks in its *outbuf* local variable waiting for transmission and it receives the token from *MEDIU*M, it transitions to state 2 with *get-tk* and then takes the *Xmit* transition to state 3, sending the first data block. Since it has one more data block to send, the next transition will be *moreD,* which will take it back to state 2. At this point the system state analysis will stop and the reachability analysis will be incomplete.

The problem can be solved by splitting the system state analysis into three parts. First, the protocol can be analyzed with no messages in the machines and the behavior of the machines including only the transitions of the token can be observed (transitions *get-tk* and *pass*). Then, the analysis can be performed with one message in the *outbuf* local variables of the machines, which allows us to analyze the transitions for receiving/ transmitting the messages in addition to the transitions including the token (*get-tk*, *Xmit*, *rcv*, *ready*, *pass-tk)*. Finally, the protocol can be analyzed with each machine having more than one message, which includes the last transition in the analysis (*moreD*). Combining the results of these parts shows that the protocol is free from deadlocks and there are no unexecuted transitions.

The *definitions* packages and the analysis results are given separately for each of the three cases outlined above in Appendix C. The system state analysis generated 16, 40 and 5 system states respectively for the parts explained above. The global analysis has generated 263 global states and there were no deadlocks or unexecuted transitions. The global reachability analysis is also given in Appendix C.

The system state analysis has reduced the number of states from 263 (global) to 61 (for all three parts). This is another example showing the advantage of the system state analysis.

VI. CONCLUSIONS AND FURTHER RESEARCH POSSIBILITIES

In this thesis, a software tool has been described which automates the analysis of protocols specified by the SCM and CFSM models. The program generates either the system state analysis or global reachability analysis for the SCM model. The program also generates the full reachability graph for a protocol specified by the CFSM model.

The major achievement of the thesis was the increase in the number of machines in the protocol specification. The previous work in [Ref. 8] was extended to allow two to eight machines in the specification. The run time and memory efficiency of the program were improved to allow the analysis of larger and more complex protocols. The user interface of the program has also been improved.

The system state analysis reduces the size of the state space greatly, but in some cases, when the system state analysis is not sufficient for the protocol analysis, the global reachability analysis is required. The *Smart Mushroom* program generates the system state graph. The *Simple* and *Big Mushroom* programs are based on exhaustive analysis, and generate the full global reachability graph. The main problem in these programs is the "state space explosion." As stated in [Ref. 16], an estimate for the maximum size of the state space that can be reached for a full reachability analysis is about 10^5 states. This is in agreement with the maximum number of states generated so far using the *Big Mushroom* program ($153565 \cong 1.53 \times 10^5$ states were generated for the example protocol described in Chapter V).

The size of the state space which can be generated is directly proportional with the memory available on the computer. For a full reachability graph, an equation can be derived for determining the maximum number of states: where,

M: Memory available on the computer (bytes).

S: Amount of memory for storing one system state (bytes).

O: Overhead (memory for storing the program and other data structures etc.).

Then, the number of states that can be analyzed is: $N = (M-O)/S$. Usually $O << M$, and O can be ignored. For instance, for the LAP-B protocol analysis described in Chapter V, M=80 MBytes, S = 516 bytes, and N = 162596. In this analysis, only 153565 states were generated by the *Simple Mushroom* program. The difference between these numbers is due to the exclusion of the overhead in the calculation. Unfortunately memory was not enough for a 100% coverage in this analysis.

In spite of the state space explosion, the programs developed in this thesis are still very helpful for analyzing protocols. A full reachability analysis may be feasible by keeping the protocol specifications as simple as possible, and using certain assumptions about the behavior of the protocol to reduce the size of the state space. For example, the size of the message queue is very important for the CFSM model. A smaller message queue decreases S and allows to analyze larger protocols. A specification with less number of processes increases the number of states that can be analyzed. Modeling the machines with less number of states is also helpful. For the SCM model, N can be increased by keeping the size of global and local variables as small as possible. A simpler protocol specification also reduces the run time.

But, in some cases, even after some simplifications, a full reachability analysis is impossible. Fortunately, still some solutions exist for the automated protocol analysis. One method which is described in [Ref. 16] is using the *supertrace* algorithm. In the *Mushroom* program, hashing is used to increase the search efficiency. In the supertrace algorithm a very large hash size (almost the whole available memory) is used, and system states are not stored. This method is explained in [Ref. 16]. For example, with a 10 MB of memory, 80 million states can be generated using this method as described in [Ref. 16]. Of course this

efficiency does not come free. Due to hash conflicts, this method cannot guarantee 100% coverage, but as a partial search technique, this algorithm is very powerful.

This thesis opens several areas for further work. One improvement would be to increase the size of the system space that can be analyzed. Adding the supertrace option to the *Mushroom* program can be a good area for further work.

The number of reachable states is usually very large and it would be awkward to print out or browse through the listing. Another improvement would be to store the reachability analysis results in the form of a database, and provide a query language that allows the user to easily analyze the results of the analysis as suggested in [Ref. 17] (for instance, querying the error sequences and certain paths between any two states etc.).

Finally, another research possibility would be to add a simulator module to the *Mushroom.* For protocols with a large size of state space, where full reachability analysis is infeasible, simulation would be useful.

The Ada programming language was used to develop *Mushroom.* Also, specification of the SCM model must be entered to the program using Ada subprograms and packages. Ada is a well-structured programming language, and supports the modular development of programs. Also, exception handling, generic units, and tasking are important features of Ada. These features were helpful in developing the program. The well-structured property of the programming language makes the input of the specification easier. The tasking mechanism of Ada would be very helpful to develop a simulator module for the program.

The *Simple Mushroom* program is used as a teaching aid in an introductory communications network course at Naval Postgraduate School. This can be another area where student can use the tool as an aid in learning the protocol design and analysis.

The *mushroom* program is a tool which it is hoped that it will greatly improve the design and analysis of protocols specified by the SCM and CFSM models. Especially, this

program may help to solve some questions concerning the SCM model which have not been completely answered.

APPENDIX A (LAP-B Protocol Information Transfer Phase)

FSM Text File

```
start
number_of_machines 6
machine 1
state 1
trans +A0 1 3
trans -D0 2 3
state 2
trans +A0 2 3
trans -D1 3 3
trans +A1 4 3
state 3
trans +A0 3 3
trans +A1 5 3
trans +A2 7 3
state 4
trans +A1 4 3
trans -D1 5 3
state 5
trans +A1 5 3
trans +A2 7 3
trans -D2 6 3
state 6
trans +A1 6 3
trans +A0 1 3
trans +A2 8 3
state 7
trans +A2 7 3
trans -D2 8 3
state 8
trans +A2 8 3
trans +A0 1 3
trans -D0 9 3
state 9
trans +A2 9 3
trans +A0 2 3
trans +A1 4 3
machine 2
state 1
trans +ENQ 4 3
trans +D0 2 3
state 2
trans +ENQ 5 3
trans +D1 3 3
state 3
trans +ENQ 6 3
trans +D2 1 3
state 4
trans -A0 1 3
state 5
trans -A1 2 3
state 6
trans -A2 3 3
machine 3
state 1
trans +D0 2 1
trans +D1 3 1
trans +D2 4 1
trans +I00 20 4
trans +I10 21 4
trans +I20 22 4
trans +I01 23 4
trans +I11 24 4
trans +I21 25 4
trans +I02 26 4
trans +I12 27 4
trans +I22 28 4
state 2
trans -ENQ 8 2
state 3
trans -ENQ 9 2
state 4
trans -ENQ 10 2
```

```
state 8
trans +A0 11 2
trans +A1 12 2
trans +A2 13 2
state 9
trans +A0 14 2
trans +A1 15 2
trans +A2 16 2
state 10
trans +A0 17 2
trans +A1 18 2
trans +A2 19 2
state 11
trans -I00 1 4
state 12
trans -I01 1 4
state 13
trans -I02 1 4
state 14
trans -I10 1 4
state 15
trans -I11 1 4
state 16
trans -I12 1 4
state 17
trans -I20 1 4
state 18
trans -I21 1 4
state 19
trans -I22 1 4
state 20
trans -D0 29 2
state 21
trans -D1 29 2
state 22
trans -D2 29 2
state 23
trans -D0 30 2
state 24
trans -D1 30 2
state 25
trans -D2 30 2
state 26
trans -D0 31 2
state 27
trans -D1 31 2
state 28
trans -D2 31 2
state 29
trans -A0 1 1
state 30
trans -A1 1 1
state 31
trans -A2 1 1
machine 4
state 1
trans +D0 2 5
trans +D1 3 5
trans +D2 4 5
trans +I00 20 3
trans +I10 21 3
trans +I20 22 3
trans +I01 23 3
trans +I11 24 3
trans +I21 25 3
trans +I02 26 3
trans +I12 27 3
trans +I22 28 3
state 2
trans -ENQ 8 6
state 3
trans -ENQ 9 6
state 4
trans -ENQ 10 6
state 8
trans +A0 11 6
trans +A1 12 6
trans +A2 13 6
```

```
state 9
trans +A0 14 6
trans +A1 15 6
trans +A2 16 6
state 10
trans +A0 17 6
trans +A1 18 6
trans +A2 19 6
state 11
trans -I00 1 3
state 12
trans -I01 1 3
state 13
trans -I02 1 3
state 14
state 15
trans -I11 1 3
state 16
trans -I12 1 3
state 17
trans -I20 1 3
trans -I10 1 3
trans -D0 9 4
state 18
trans -I21 1 3
state 19
trans -I22 1 3
state 20
trans -D0 29 6
state 21
trans -D1 29 6
state 22
trans -D2 29 6
state 23
trans -D0 30 6
state 24
trans -D1 30 6
state 25
trans -D2 30 6
state 26
trans -D0 31 6
state 27
trans -D1 31 6
state 28
trans -D2 31 6
state 29
trans -A0 1 5
state 30
trans -A1 1 5
state 31
trans -A2 1 5
machine 5
state 1
trans +A0 1 4
trans -D0 2 4
state 2
trans +A0 2 4
trans -D1 3 4
trans +A1 4 4
state 3
trans +A0 3 4
trans +A1 5 4
trans +A2 7 4
state 4
trans +A1 4 4
trans -D1 5 4
state 5
trans +A1 5 4
trans +A2 7 4
trans -D2 6 4
state 6
trans +A1 6 4
trans +A0 1 4
trans +A2 8 4
state 7
trans +A2 7 4
trans -D2 8 4
```

```
state 8
trans +A2 8 4
trans +A0 1 4
trans -D0 9 4
state 9
trans +A2 9 4
trans +A0 2 4
trans +A1 4 4
machine 6
state 1
trans +ENQ 4 4
trans +D0 2 4
state 2
trans +ENQ 5 4
trans +D1 3 4
state 3
trans +ENQ 6 4
trans +D2 1 4
state 4
trans -A0 1 4
state 5
trans -A1 2 4
state 6
trans -A2 3 4
initial_state 1 1 1 1 1 1
finish
```

Program Output

REACHABILITY ANALYSIS of : fad.fsm
SPECIFICATION

Machine 1 State Transitions			
From	To	other machine	Transition
1	1	3	r A0
1	2	3	s D0
2	2	3	r A0
2	3	3	s D1
2	4	3	r A1
3	3	3	r A0
3	5	3	r A1
3	7	3	r A2
4	4	3	r A1
4	5	3	s D1
5	5	3	r A1
5	7	3	r A2
5	6	3	s D2
6	6	3	r A1
6	1	3	r A0
6	8	3	r A2
7	7	3	r A2
7	8	3	s D2
8	8	3	r A2
8	1	3	r A0
8	9	3	s D0
9	9	3	r A2
9	2	3	r A0
9	4	3	r A1

Machine 2 State Transitions			
From	To	other machine	Transition
1	4	3	r ENQ
1	2	3	r D0
2	5	3	r ENQ
2	3	3	r D1
3	6	3	r ENQ
3	1	3	r D2
4	1	3	s A0
5	2	3	s A1
6	3	3	s A2

Machine 6 State Transitions			
From	To	other machine	Transition
1	4	4	r ENQ
1	2	4	r D0
2	5	4	r ENQ
2	3	4	r D1
3	6	4	r ENQ
3	1	4	r D2
4	1	4	s A0
5	2	4	s A1
6	3	4	s A2

REACHABILITY GRAPH

```
1 [ 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]
  -D0 3 [ 2,E,D0 ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E,1,E,E,E,E,E]      2
  -D0 4 [ 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,D0,E, 1,E,E,E,E,E]      3

2 [ 2,E,D0 ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]
  -D1 3 [ 3,E,D0 D1 ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]      4
  +D0 1 [ 2,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]      5
  -D0 4 [ 2,E,D0 ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,D0 ,E, 1,E,E,E,E,E]      6

3 [ 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,D0 ,E, 1,E,E,E,E,E]
  -D0 3 [ 2,E,D0 ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,D0 ,E, 1,E,E,E,E,E]      6
  +D0 5 [ 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,E,E, 2,E,E,E,E,E, 1,E,E,E,E,E]      7
  -D1 4 [ 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 3,E,E,E,D0 D1 ,E, 1,E,E,E,E,E]      8

4 [ 3,E,D0 D1 ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]
  +D0 1 [ 3,E,D1 ,E,E,E, 1,E,E,E,E,E, 2,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]      9
  -D0 4 [ 3,E,D0 D1 ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,D0 ,E, 1,E,E,E,E,E]      10

5 [ 2,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]
  -D1 3 [ 3,E,D1 ,E,E,E, 1,E,E,E,E,E, 2,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]      9
  -ENQ 2 [ 2,E,E,E,E,E, 1,E,E,E,E,E, 8,E,ENQ ,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E]      11
  -D0 4 [ 2,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,E,E, 1,E,E,E,E,E, 2,E,E,E,D0,E, 1,E,E,E,E,E]      12
```

.

.

```
17034 [ 3,E,E,E,E,E, 3,E,E,E,E,E, 1,E,E,E,E,E, 1,E,E,E,E,E, 3,E,E,E,E,E, 3,E,E,E,E,E]
                        **********DEADLOCK condition**************

17035 [ 6,E,E,E,E,E, 3,E,E,E,E,E, 30,E,E, 111 121,E,E, 1,E,E,E,E,E, 3,E,E,E,E,E, 2,E,E,E,E,E]
   -A1 1 [ 6,E,E,E,E,E, 3,E,E,E,E,E, 1,A1 ,E,111121,E,E, 1,E,E,E,E,E, 3,E,E,E,E,E,2,E,E,E,E,E]        17034
```

.

.

73391. . .

SUMMARY OF REACHABILTY ANALYSIS (ANALYSIS COMPLETED)

Total number of states generated : 73391
Number of states analyzed : 73391
number of deadlocks : 1
number of unspecified receptions : 0
maximum message queue size : 6
channel overflow : NONE

UNEXECUTED TRANSITIONS
****NONE****

APPENDIX B (Go back N Window Size of 10)

FSM Text File

```
start
number_of_machines 2
machine 1
state 0
trans snd_data 1
state 1
trans rcv_ack0 0
trans snd_data 2
state 2
trans rcv_ack0 0
trans rcv_ack1 1
trans snd_data 3
state 3
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans snd_data 4
state 4
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
trans snd_data 5
state 5
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
trans rcv_ack4 4
trans snd_data 6
state 6
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
trans rcv_ack4 4
trans rcv_ack5 5
trans snd_data 7
state 7
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
trans rcv_ack4 4
trans rcv_ack5 5
trans rcv_ack6 6
trans snd_data 8
state 8
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
trans rcv_ack4 4
trans rcv_ack5 5
trans rcv_ack6 6
trans rcv_ack7 7
trans snd_data 9
state 9
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
trans rcv_ack4 4
trans rcv_ack5 5
trans rcv_ack6 6
trans rcv_ack7 7
trans rcv_ack8 8
trans snd_data 10
```

```
state 10
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
trans rcv_ack4 4
trans rcv_ack5 5
trans rcv_ack6 6
trans rcv_ack7 7
trans rcv_ack8 8
trans rcv_ack9 9
machine 2
state 0
trans rcv_data 1
state 1
trans rcv_data 2
trans snd_ack 0
state 2
trans rcv_data 3
trans snd_ack 0
state 3
trans rcv_data 4
trans snd_ack 0
state 4
trans rcv_data 5
trans snd_ack 0
state 5
trans rcv_data 6
trans snd_ack 0
state 6
trans rcv_data 7
trans snd_ack 0
state 7
trans rcv_data 8
trans snd_ack 0
state 8
trans rcv_data 9
trans snd_ack 0
state 9
trans rcv_data 10
trans snd_ack 0
state 10
trans snd_ack 0
initial_state 0 0
finish
```

Variable Definitions

```
with TEXT_IO; use TEXT_IO;
package definitions is
   num_of_machines : constant := 2;
   type scm_transition_type is
(snd_data,rcv_data,rcv_ack0,rcv_ack1,rcv_ack2,rcv_ack3,rcv_ack4,
         rcv_ack5,rcv_ack6,rcv_ack7,rcv_ack8,rcv_ack9,snd_ack,unused);

   type buffer_type   is (d0,d1,d2,d3,d4,d5,d6,d7,d8,d9,e);
   package buff_enum_io is new enumeration_io (buffer_type);
   use buff_enum_io;
   type buffer_array_type is array(1..10) of buffer_type;
   type seq_array_type is array(1..10) of integer range -1..10;

   type machine1_state_type is
      record
        Sdata :buffer_array_type := (d0,d1,d2,d3,d4,d5,d6,d7,d8,d9);
        seq   : integer range 0..10 := 0;
        i      :integer range 1..10 := 1;
   end record;

   type dummy_type is range 1..255;

   type machine2_state_type is
      record
        Rdata:buffer_type := e;
        exp   :integer range 0..10 := 0;
        j     :integer range 1..10 := 1;
   end record;
   type machine3_state_type is
    record
      dummy : dummy_type;
    end record;

   type machine4_state_type is
     record
       dummy : dummy_type;
     end record;

   type machine5_state_type is
     record
      dummy : dummy_type;
     end record;

   type machine6_state_type is
     record
      dummy : dummy_type;
     end record;

   type machine7_state_type is
     record
      dummy : dummy_type;
     end record;

   type machine8_state_type is
     record
      dummy : dummy_type;
     end record;

   type global_variable_type is
      record
        DATA : buffer_array_type   := (e,e,e,e,e,e,e,e,e,e);
        SEQ   : seq_array_type      := (-1,-1,-1,-1,-1,-1,-1,-1,-1,-1);
        ACK   : integer range -1..10 := -1;
      end record;

end definitions;
```

Predicate-action Table

```
separate (main)
procedure  Analyze_Predicates_Machine1(local : machine1_state_type;
                                       GLOBAL: global_variable_type;
                                       s : natural;
                                       w :in out transition_stack_package.stack) is
      temp1  : integer := GLOBAL.ACK + 0;
      temp2  : integer := (GLOBAL.ACK + 1) mod 11;
      temp3  : integer := (GLOBAL.ACK + 2) mod 11;
      temp4  : integer := (GLOBAL.ACK + 3) mod 11;
      temp5  : integer := (GLOBAL.ACK + 4) mod 11;
      temp6  : integer := (GLOBAL.ACK + 5) mod 11;
      temp7  : integer := (GLOBAL.ACK + 6) mod 11;
      temp8  : integer := (GLOBAL.ACK + 7) mod 11;
      temp9  : integer := (GLOBAL.ACK + 8) mod 11;
      temp10 : integer := (GLOBAL.ACK + 9) mod 11;



begin
  case s is
    when 0 =>
      if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w,snd_data);
      end if;
    when 1 =>
      if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w,snd_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack0);
      end if;
    when 2 =>
      if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w,snd_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack0);
      end if;
      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack1);
      end if;
    when 3 =>
      if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w,snd_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack0);
      end if;
      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack1);
      end if;
      if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack2);
      end if;
    when 4 =>
      if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w,snd_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack0);
      end if;
      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack1);
      end if;
```

```
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;
  when 5 =>
    if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
      Push(w,snd_data);
    end if;

    if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;

    if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack4);
    end if;
  when 6 =>
    if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
      Push(w,snd_data);
    end if;

    if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;

    if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack4);
    end if;
    if ((temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack5);
    end if;
  when 7 =>
    if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
      Push(w,snd_data);
    end if;

    if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;

    if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack4);
    end if;
    if ((temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack5);
    end if;
    if ((temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack6);
    end if;
  when 8 =>

    if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
```

```
      Push(w,snd_data);
    end if;

    if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;

    if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack4);
    end if;
    if ((temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack5);
    end if;
    if ((temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack6);
    end if;
    if ((temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack7);
    end if;

   when 9 =>
    if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
      Push(w,snd_data);
    end if;

    if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;

    if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack4);
    end if;
    if ((temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack5);
    end if;
    if ((temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack6);
    end if;
    if ((temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack7);
    end if;
    if ((temp9 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack8);
    end if;
    if ((temp10 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack9);
    end if;

  when 10 =>

    if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack1);
    end if;
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack2);
    end if;
    if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
      Push(w,rcv_ack3);
    end if;
    if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
```

```
        Push(w,rcv_ack4);
      end if;
      if ((temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack5);
      end if;
      if ((temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack6);
      end if;
      if ((temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack7);
      end if;
      if ((temp9 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack8);
      end if;
      if ((temp10 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w,rcv_ack9);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine1;
------------------------------------------------------------------------------------------------
separate (main)
procedure Analyze_Predicates_Machine2(local : machine2_state_type;
                                      GLOBAL: global_variable_type;
                                      s: natural;
                                      w :in out transition_stack_package.stack) is
begin
  case s is
    when 0  =>
      if ((GLOBAL.DATA(local.j)/=E) and (GLOBAL.SEQ(local.j) = local.exp)) then
        Push(w,rcv_data);
      end if;
    when 1|2|3|4|5|6|7|8|9 =>
      if (GLOBAL.DATA(local.j)=E) then
       Push(w,snd_ack);
      end if;
      if ((GLOBAL.DATA(local.j)/=E) and (GLOBAL.SEQ(local.j) = local.exp)) then
        Push(w,rcv_data);
      end if;
    when 10 =>
      if (GLOBAL.DATA(local.j)=E) then
       Push(w,snd_ack);
      end if;

    when others =>
      null;
  end case;
end Analyze_Predicates_Machine2;
------------------------------------------------------------------------------------------------
separate (main)
procedure Analyze_Predicates_Machine3(local : machine3_state_type;
                                      GLOBAL: global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is

begin
  null;
end Analyze_Predicates_Machine3;
------------------------------------------------------------------------------------------------
separate (main)
procedure Analyze_Predicates_Machine4(local : machine4_state_type;
                                      GLOBAL: global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is

begin
  null;
end Analyze_Predicates_Machine4;
------------------------------------------------------------------------------------------------
separate (main)
procedure Analyze_Predicates_Machine5(local : machine5_state_type;
                                      GLOBAL: global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is

begin
  null;
end Analyze_Predicates_Machine5;

------------------------------------------------------------------------------------------------
```

```
separate (main)
procedure Analyze_Predicates_Machine6(local : machine6_state_type;
                                      GLOBAL: global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is

begin
  null;
end Analyze_Predicates_Machine6;

-------------------------------------------------------------------------------------------------------
separate (main)
procedure Analyze_Predicates_Machine7(local : machine7_state_type;
                                      GLOBAL: global_variable_type;
                                      s : natural;
                                      w : in out
transition_stack_package.stack) is

begin
  null;
end Analyze_Predicates_Machine7;

-------------------------------------------------------------------------------------------------------
separate (main)
procedure Analyze_Predicates_Machine8(local : machine8_state_type;
                                      GLOBAL: global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is

begin
  null;
end Analyze_Predicates_Machine8;

-------------------------------------------------------------------------------------------------------

separate (main)
procedure Action(in_system_state  : in out Gstate_record_type;
                 in_transition    : in out scm_transition_type;
                 out_system_state : in out Gstate_record_type) is


begin
   case (in_transition) is
      when snd_data =>

         out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine1_state.i) :=
          in_system_state.machine1_state.Sdata(in_system_state.machine1_state.i);
         out_system_state.GLOBAL_VARIABLES.SEQ(in_system_state.machine1_state.i) :=
                                              in_system_state.machine1_state.seq;
         out_system_state.machine1_state.i := (in_system_state.machine1_state.i mod 10) + 1;
         out_system_state.machine1_state.seq := (((in_system_state.machine1_state.seq) + 1)mod 11);

      when rcv_ack0 | rcv_ack1 | rcv_ack2 | rcv_ack3 | rcv_ack4
            | rcv_ack5 |rcv_ack6 | rcv_ack7 |rcv_ack8|rcv_ack9 =>

            out_system_state.GLOBAL_VARIABLES.ACK := -1;

      when snd_ack =>

            out_system_state.GLOBAL_VARIABLES.ACK := in_system_state.machine2_state.exp;
            out_system_state.machine2_state.Rdata := e;

      when rcv_data =>

            out_system_state.machine2_state.Rdata  :=
                    in_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine2_state.j);
            out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine2_state.j) := E;
            out_system_state.GLOBAL_VARIABLES.SEQ (in_system_state.machine2_state.j) := -1;
            out_system_state.machine2_state.j   := (in_system_state.machine2_state.j mod 10) + 1;
            out_system_state.machine2_state.exp := (((in_system_state.machine2_state.exp) + 1)mod 11);
      when others =>
            put_line("There is an error in the Action procedure");
    end case;
end Action;
```

Output Format

```
separate (main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
   if print_header then
     new_line(2);
     set_col(7);
     put_line("   m1(seq,i,Sdata), m2(exp,j,Rdata), (DATA,SEQ,ACK)");
     print_header := false;
   else
     put("   [" & integer'image(tuple.machine_state(1)) );
     put(" , ");
     put(tuple.machine1_state.seq, width => 1);
     put(" , ");
     put(tuple.machine1_state.i, width => 1);
     put(" , ");
     buff_enum_io.put(tuple.machine1_state.Sdata(1),set => upper_case);
     put(" ," & integer'image(tuple.machine_state(2)) );
     put(" , ");
     put(tuple.machine2_state.exp, width => 1);
     put(" , ");
     put(tuple.machine2_state.j, width => 1);
     put(" , ");
     buff_enum_io.put(tuple.machine2_state.Rdata,set => upper_case);
     for i in 1..10 loop
       put(" , ");
       buff_enum_io.put(tuple.GLOBAL_VARIABLES.DATA(i),set => upper_case);
       put(",");
       put(tuple.GLOBAL_VARIABLES.SEQ(i),width=>1);
     end loop;
     put(" ,");
     put(tuple.GLOBAL_VARIABLES.ACK, width => 1);
     put(" ]");
   end if;

end output_Gtuple;
```

Program Output (System State Analysis)

REACHABILITY ANALYSIS of :gbn_10.scm
SPECIFICATION

Machine 1 State Transitions

From	To	Transition
0	1	snd_data
1	0	rcv_ack0
1	2	snd_data
2	0	rcv_ack0
2	1	rcv_ack1
2	3	snd_data
3	0	rcv_ack0
3	1	rcv_ack1
3	2	rcv_ack2
3	4	snd_data
4	0	rcv_ack0
4	1	rcv_ack1
4	2	rcv_ack2
4	3	rcv_ack3
4	5	snd_data
5	0	rcv_ack0
5	1	rcv_ack1
5	2	rcv_ack2
5	3	rcv_ack3
5	4	rcv_ack4
5	6	snd_data
6	0	rcv_ack0
6	1	rcv_ack1
6	2	rcv_ack2
6	3	rcv_ack3
6	4	rcv_ack4
6	5	rcv_ack5
6	7	snd_data
7	0	rcv_ack0
7	1	rcv_ack1
7	2	rcv_ack2
7	3	rcv_ack3
7	4	rcv_ack4
7	5	rcv_ack5
7	6	rcv_ack6
7	8	snd_data
8	0	rcv_ack0
8	1	rcv_ack1
8	2	rcv_ack2
8	3	rcv_ack3
8	4	rcv_ack4
8	5	rcv_ack5
8	6	rcv_ack6
8	7	rcv_ack7
8	9	snd_data
9	0	rcv_ack0
9	1	rcv_ack1
9	2	rcv_ack2
9	3	rcv_ack3
9	4	rcv_ack4
9	5	rcv_ack5
9	6	rcv_ack6
9	7	rcv_ack7
9	8	rcv_ack8
9	10	snd_data
10	0	rcv_ack0
10	1	rcv_ack1
10	2	rcv_ack2
10	3	rcv_ack3
10	4	rcv_ack4
10	5	rcv_ack5
10	6	rcv_ack6
10	7	rcv_ack7
10	8	rcv_ack8
10	9	rcv_ack9

Machine 2 State Transitions		
From	To	Transition
0	1	rcv_data
1	2	rcv_data
1	0	snd_ack
2	3	rcv_data
2	0	snd_ack
3	4	rcv_data
3	0	snd_ack
4	5	rcv_data
4	0	snd_ack
5	6	rcv_data
5	0	snd_ack
6	7	rcv_data
6	0	snd_ack
7	8	rcv_data
7	0	snd_ack
8	9	rcv_data
8	0	snd_ack
9	10	rcv_data
9	0	snd_ack
10	0	snd_ack

REACHABILITY GRAPH

0	[0, 0] 0	snd_data	1
1	[1, 0] 0	snd_data	2
		rcv_data	3
2	[2, 0] 0	snd_data	4
		rcv_data	5
3	[1, 1] 0	snd_data	5
		snd_ack	6
4	[3, 0] 0	snd_data	7
		rcv_data	8
5	[2, 1] 0	snd_data	8
		rcv_data	9
6	[1, 0] 1	rcv_ack0	0
		snd_data	10
7	[4, 0] 0	snd_data	11
		rcv_data	12
8	[3, 1] 0	snd_data	12
		rcv_data	13
9	[2, 2] 0	snd_data	13
		snd_ack	14
10	[2, 0] 1	rcv_ack1	1
		snd_data	15
		rcv_data	16
11	[5, 0] 0	snd_data	17
		rcv_data	18
12	[4, 1] 0	snd_data	18
		rcv_data	19
13	[3, 2] 0	snd_data	19
		rcv_data	20
14	[2, 0] 2	rcv_ack0	0
		snd_data	21
15	[3, 0] 1	rcv_ack2	2
		snd_data	22
		rcv_data	23
16	[2, 1] 1	rcv_ack1	3
		snd_data	23
		snd_ack	14
17	[6, 0] 0	snd_data	24
		rcv_data	25
18	[5, 1] 0	snd_data	25
		rcv_data	26
19	[4, 2] 0	snd_data	26
		rcv_data	27

```
20 [ 3, 3 ] 0        snd_data  27
                     snd_ack   28
21 [ 3, 0 ] 2        rcv_ack1   1
                     snd_data  29
                     rcv_data  30
22 [ 4, 0 ] 1        rcv_ack3   4
                     snd_data  31
                     rcv_data  32
23 [ 3, 1 ] 1        rcv_ack2   5
                     snd_data  32
                     rcv_data  33
24 [ 7, 0 ] 0        snd_data  34
                     rcv_data  35
25 [ 6, 1 ] 0        snd_data  35
                     rcv_data  36
26 [ 5, 2 ] 0        snd_data  36
                     rcv_data  37
27 [ 4, 3 ] 0        snd_data  37
                     rcv_data  38
28 [ 3, 0 ] 3        rcv_ack0   0
                     snd_data  39
29 [ 4, 0 ] 2        rcv_ack2   2
                     snd_data  40
                     rcv_data  41
30 [ 3, 1 ] 2        rcv_ack1   3
                     snd_data  41
                     snd_ack   28
31 [ 5, 0 ] 1        rcv_ack4   7
                     snd_data  42
                     rcv_data  43
32 [ 4, 1 ] 1        rcv_ack3   8
                     snd_data  43
                     rcv_data  44
33 [ 3, 2 ] 1        rcv_ack2   9
                     snd_data  44
                     snd_ack   28
34 [ 8, 0 ] 1        snd_data  45
                     rcv_data  46
35 [ 7, 1 ] 0        snd_data  46
                     rcv_data  47
36 [ 6, 2 ] 0        snd_data  47
                     rcv_data  48
37 [ 5, 3 ] 0        snd_data  48
                     rcv_data  49
38 [ 4, 4 ] 0        snd_data  49
                     snd_ack   50
39 [ 4, 0 ] 3        rcv_ack1   1
                     snd_data  51
                     rcv_data  52
40 [ 5, 0 ] 2        rcv_ack3   4
                     snd_data  53
                     rcv_data  54
41 [ 4, 1 ] 2        rcv_ack2   5
                     snd_data  54
                     rcv_data  55
42 [ 6, 0 ] 1        rcv_ack5  11
                     snd_data  56
                     rcv_data  57
43 [ 5, 1 ] 1        rcv_ack4  12
                     snd_data  57
                     rcv_data  58
44 [ 4, 2 ] 1        rcv_ack3  13
                     snd_data  58
                     rcv_data  59
45 [ 9, 0 ] 2        snd_data  60
                     rcv_data  61
46 [ 8, 1 ] 0        snd_data  61
                     rcv_data  62
47 [ 7, 2 ] 0        snd_data  62
                     rcv_data  63
48 [ 6, 3 ] 0        snd_data  63
                     rcv_data  64
49 [ 5, 4 ] 0        snd_data  64
```

```
                         rcv_data  65
50 [ 4, 0 ] 4            rcv_ack0   0
                         snd_data  66
51 [ 5, 0 ] 3            rcv_ack2   2
                         snd_data  67
                         rcv_data  68
52 [ 4, 1 ] 3            rcv_ack1   3
                         snd_data  68
                         snd_ack   50
53 [ 6, 0 ] 2            rcv_ack4   7
                         snd_data  69
                         rcv_data  70
54 [ 5, 1 ] 2            rcv_ack3   8
                         snd_data  70
                         rcv_data  71
55 [ 4, 2 ] 2            rcv_ack2   9
                         snd_data  71
                         snd_ack   50
56 [ 7, 0 ] 1            rcv_ack6  17
                         snd_data  72
                         rcv_data  73
57 [ 6, 1 ] 1            rcv_ack5  18
                         snd_data  73
                         rcv_data  74
58 [ 5, 2 ] 1            rcv_ack4  19
                         snd_data  74
                         rcv_data  75
59 [ 4, 3 ] 1            rcv_ack3  20
                         snd_data  75
                         snd_ack   50
60 [10, 0 ] 3            rcv_data  76
61 [ 9, 1 ] 1            snd_data  76
                         rcv_data  77
62 [ 8, 2 ] 0            snd_data  77
                         rcv_data  78
63 [ 7, 3 ] 0            snd_data  78
                         rcv_data  79
64 [ 6, 4 ] 0            snd_data  79
                         rcv_data  80
65 [ 5, 5 ] 0            snd_data  80
                         snd_ack   81
66 [ 5, 0 ] 4            rcv_ack1   1
                         snd_data  82
                         rcv_data  83
67 [ 6, 0 ] 3            rcv_ack3   4
                         snd_data  84
                         rcv_data  85
68 [ 5, 1 ] 3            rcv_ack2   5
                         snd_data  85
                         rcv_data  86
69 [ 7, 0 ] 2            rcv_ack5  11
                         snd_data  87
                         rcv_data  88
70 [ 6, 1 ] 2            rcv_ack4  12
                         snd_data  88
                         rcv_data  89
71 [ 5, 2 ] 2            rcv_ack3  13
                         snd_data  89
                         rcv_data  90
72 [ 8, 0 ] 2            rcv_ack7  24
                         snd_data  91
                         rcv_data  92
73 [ 7, 1 ] 1            rcv_ack6  25
                         snd_data  92
                         rcv_data  93
74 [ 6, 2 ] 1            rcv_ack5  26
                         snd_data  93
                         rcv_data  94
75 [ 5, 3 ] 1            rcv_ack4  27
                         snd_data  94
                         rcv_data  95
76 [10, 1 ] 2            rcv_data  96
77 [ 9, 2 ] 0            snd_data  96
```

```
                rcv_data  97
 78 [ 8, 3 ] 0  snd_data  97
                rcv_data  98
 79 [ 7, 4 ] 0  snd_data  98
                rcv_data  99
 80 [ 6, 5 ] 0  snd_data  99
                rcv_data 100
 81 [ 5, 0 ] 5  rcv_ack0   0
                snd_data 101
 82 [ 6, 0 ] 4  rcv_ack2   2
                snd_data 102
                rcv_data 103
 83 [ 5, 1 ] 4  rcv_ack1   3
                snd_data 103
                snd_ack   81
 84 [ 7, 0 ] 3  rcv_ack4   7
                snd_data 104
                rcv_data 105
 85 [ 6, 1 ] 3  rcv_ack3   8
                snd_data 105
                rcv_data 106
 86 [ 5, 2 ] 3  rcv_ack2   9
                snd_data 106
                snd_ack   81
 87 [ 8, 0 ] 3  rcv_ack6  17
                snd_data 107
                rcv_data 108
 88 [ 7, 1 ] 2  rcv_ack5  18
                snd_data 108
                rcv_data 109
 89 [ 6, 2 ] 2  rcv_ack4  19
                snd_data 109
                rcv_data 110
 90 [ 5, 3 ] 2  rcv_ack3  20
                snd_data 110
                snd_ack   81
 91 [ 9, 0 ] 3  rcv_ack8  34
                snd_data 111
                rcv_data 112
 92 [ 8, 1 ] 1  rcv_ack7  35
                snd_data 112
                rcv_data 113
 93 [ 7, 2 ] 1  rcv_ack6  36
                snd_data 113
                rcv_data 114
 94 [ 6, 3 ] 1  rcv_ack5  37
                snd_data 114
                rcv_data 115
 95 [ 5, 4 ] 1  rcv_ack4  38
                snd_data 115
                snd_ack   81
 96 [10, 2 ] 1  rcv_data 116
 97 [ 9, 3 ] 0  snd_data 116
                rcv_data 117
 98 [ 8, 4 ] 0  snd_data 117
                rcv_data 118
 99 [ 7, 5 ] 0  snd_data 118
                rcv_data 119
100 [ 6, 6 ] 0  snd_data 119
                snd_ack  120
101 [ 6, 0 ] 5  rcv_ack1   1
                snd_data 121
                rcv_data 122
102 [ 7, 0 ] 4  rcv_ack3   4
                snd_data 123
                rcv_data 124
103 [ 6, 1 ] 4  rcv_ack2   5
                snd_data 124
                rcv_data 125
104 [ 8, 0 ] 4  rcv_ack5  11
                snd_data 126
                rcv_data 127
105 [ 7, 1 ] 3  rcv_ack4  12
```

```
                        snd_data 127
                        rcv_data 128
106 [ 6, 2 ] 3          rcv_ack3  13
                        snd_data 128
                        rcv_data 129
107 [ 9, 0 ] 4          rcv_ack7  24
                        snd_data 130
                        rcv_data 131
108 [ 8, 1 ] 2          rcv_ack6  25
                        snd_data 131
                        rcv_data 132
109 [ 7, 2 ] 2          rcv_ack5  26
                        snd_data 132
                        rcv_data 133
110 [ 6, 3 ] 2          rcv_ack4  27
                        snd_data 133
                        rcv_data 134
111 [10, 0 ] 4          rcv_ack9  45
                        rcv_data 135
112 [ 9, 1 ] 2          rcv_ack8  46
                        snd_data 135
                        rcv_data 136
113 [ 8, 2 ] 1          rcv_ack7  47
                        snd_data 136
                        rcv_data 137
114 [ 7, 3 ] 1          rcv_ack6  48
                        snd_data 137
                        rcv_data 138
115 [ 6, 4 ] 1          rcv_ack5  49
                        snd_data 138
                        rcv_data 139
116 [10, 3 ] 0          rcv_data 140
117 [ 9, 4 ] 0          snd_data 140
                        rcv_data 141
118 [ 8, 5 ] 0          snd_data 141
                        rcv_data 142
119 [ 7, 6 ] 0          snd_data 142
                        rcv_data 143
120 [ 6, 0 ] 6          rcv_ack0   0
                        snd_data 144
121 [ 7, 0 ] 5          rcv_ack2   2
                        snd_data 145
                        rcv_data 146
122 [ 6, 1 ] 5          rcv_ack1   3
                        snd_data 146
                        snd_ack  120
123 [ 8, 0 ] 5          rcv_ack4   7
                        snd_data 147
                        rcv_data 148
124 [ 7, 1 ] 4          rcv_ack3   8
                        snd_data 148
                        rcv_data 149
125 [ 6, 2 ] 4          rcv_ack2   9
                        snd_data 149
                        snd_ack  120
126 [ 9, 0 ] 5          rcv_ack6  17
                        snd_data 150
                        rcv_data 151
127 [ 8, 1 ] 3          rcv_ack5  18
                        snd_data 151
                        rcv_data 152
128 [ 7, 2 ] 3          rcv_ack4  19
                        snd_data 152
                        rcv_data 153
129 [ 6, 3 ] 3          rcv_ack3  20
                        snd_data 153
                        snd_ack  120
130 [10, 0 ] 5          rcv_ack8  34
                        rcv_data 154
131 [ 9, 1 ] 3          rcv_ack7  35
                        snd_data 154
                        rcv_data 155
132 [ 8, 2 ] 2          rcv_ack6  36
```

```
                        snd_data 155
                        rcv_data 156
133 [ 7, 3 ] 2          rcv_ack5  37
                        snd_data 156
                        rcv_data 157
134 [ 6, 4 ] 2          rcv_ack4  38
                        snd_data 157
                        snd_ack  120
135 [10, 1 ] 3          rcv_ack9  61
                        rcv_data 158
136 [ 9, 2 ] 1          rcv_ack8  62
                        snd_data 158
                        rcv_data 159
137 [ 8, 3 ] 1          rcv_ack7  63
                        snd_data 159
                        rcv_data 160
138 [ 7, 4 ] 1          rcv_ack6  64
                        snd_data 160
                        rcv_data 161
139 [ 6, 5 ] 1          rcv_ack5  65
                        snd_data 161
                        snd_ack  120
140 [10, 4 ] 0          rcv_data 162
141 [ 9, 5 ] 0          snd_data 162
                        rcv_data 163
142 [ 8, 6 ] 0          snd_data 163
                        rcv_data 164
143 [ 7, 7 ] 0          snd_data 164
                        snd_ack  165
144 [ 7, 0 ] 6          rcv_ack1   1
                        snd_data 166
                        rcv_data 167
145 [ 8, 0 ] 6          rcv_ack3   4
                        snd_data 168
                        rcv_data 169
146 [ 7, 1 ] 5          rcv_ack2   5
                        snd_data 169
                        rcv_data 170
147 [ 9, 0 ] 6          rcv_ack5  11
                        snd_data 171
                        rcv_data 172
148 [ 8, 1 ] 4          rcv_ack4  12
                        snd_data 172
                        rcv_data 173
149 [ 7, 2 ] 4          rcv_ack3  13
                        snd_data 173
                        rcv_data 174
150 [10, 0 ] 6          rcv_ack7  24
                        rcv_data 175
151 [ 9, 1 ] 4          rcv_ack6  25
                        snd_data 175
                        rcv_data 176
152 [ 8, 2 ] 3          rcv_ack5  26
                        snd_data 176
                        rcv_data 177
153 [ 7, 3 ] 3          rcv_ack4  27
                        snd_data 177
                        rcv_data 178
154 [10, 1 ] 4          rcv_ack8  46
                        rcv_data 179
155 [ 9, 2 ] 2          rcv_ack7  47
                        snd_data 179
                        rcv_data 180
156 [ 8, 3 ] 2          rcv_ack6  48
                        snd_data 180
                        rcv_data 181
157 [ 7, 4 ] 2          rcv_ack5  49
                        snd_data 181
                        rcv_data 182
158 [10, 2 ] 2          rcv_ack9  77
                        rcv_data 183
159 [ 9, 3 ] 1          rcv_ack8  78
                        snd_data 183
```

```
                        rcv_data 184
160 [ 8, 4 ] 1          rcv_ack7  79
                        snd_data 184
                        rcv_data 185
161 [ 7, 5 ] 1          rcv_ack6  80
                        snd_data 185
                        rcv_data 186
162 [10, 5 ] 0          rcv_data 187
163 [ 9, 6 ] 0          snd_data 187
                        rcv_data 188
164 [ 8, 7 ] 0          snd_data 188
                        rcv_data 189
165 [ 7, 0 ] 7          rcv_ack0   0
                        snd_data 190
166 [ 8, 0 ] 7          rcv_ack2   2
                        snd_data 191
                        rcv_data 192
167 [ 7, 1 ] 6          rcv_ack1   3
                        snd_data 192
                        snd_ack  165
168 [ 9, 0 ] 7          rcv_ack4   7
                        snd_data 193
                        rcv_data 194
169 [ 8, 1 ] 5          rcv_ack3   8
                        snd_data 194
                        rcv_data 195
170 [ 7, 2 ] 5          rcv_ack2   9
                        snd_data 195
                        snd_ack  165
171 [10, 0 ] 7          rcv_ack6  17
                        rcv_data 196
172 [ 9, 1 ] 5          rcv_ack5  18
                        snd_data 196
                        rcv_data 197
173 [ 8, 2 ] 4          rcv_ack4  19
                        snd_data 197
                        rcv_data 198
174 [ 7, 3 ] 4          rcv_ack3  20
                        snd_data 198
                        snd_ack  165
175 [10, 1 ] 5          rcv_ack7  35
                        rcv_data 199
176 [ 9, 2 ] 3          rcv_ack6  36
                        snd_data 199
                        rcv_data 200
177 [ 8, 3 ] 3          rcv_ack5  37
                        snd_data 200
                        rcv_data 201
178 [ 7, 4 ] 3          rcv_ack4  38
                        snd_data 201
                        snd_ack  165
179 [10, 2 ] 3          rcv_ack8  62
                        rcv_data 202
180 [ 9, 3 ] 2          rcv_ack7  63
                        snd_data 202
                        rcv_data 203
181 [ 8, 4 ] 2          rcv_ack6  64
                        snd_data 203
                        rcv_data 204
182 [ 7, 5 ] 2          rcv_ack5  65
                        snd_data 204
                        snd_ack  165
183 [10, 3 ] 1          rcv_ack9  97
                        rcv_data 205
184 [ 9, 4 ] 1          rcv_ack8  98
                        snd_data 205
                        rcv_data 206
185 [ 8, 5 ] 1          rcv_ack7  99
                        snd_data 206
                        rcv_data 207
186 [ 7, 6 ] 1          rcv_ack6 100
                        snd_data 207
                        snd_ack  165
```

```
187 [10, 6 ] 0      rcv_data 208
188 [ 9, 7 ] 0      snd_data 208
                    rcv_data 209
189 [ 8, 8 ] 0      snd_data 209
                    snd_ack  210
190 [ 8, 0 ] 8      rcv_ack1   1
                    snd_data 211
                    rcv_data 212
191 [ 9, 0 ] 8      rcv_ack3   4
                    snd_data 213
                    rcv_data 214
192 [ 8, 1 ] 6      rcv_ack2   5
                    snd_data 214
                    rcv_data 215
193 [10, 0 ] 8      rcv_ack5  11
                    rcv_data 216
194 [ 9, 1 ] 6      rcv_ack4  12
                    snd_data 216
                    rcv_data 217
195 [ 8, 2 ] 5      rcv_ack3  13
                    snd_data 217
                    rcv_data 218
196 [10, 1 ] 6      rcv_ack6  25
                    rcv_data 219
197 [ 9, 2 ] 4      rcv_ack5  26
                    snd_data 219
                    rcv_data 220
198 [ 8, 3 ] 4      rcv_ack4  27
                    snd_data 220
                    rcv_data 221
199 [10, 2 ] 4      rcv_ack7  47
                    rcv_data 222
200 [ 9, 3 ] 3      rcv_ack6  48
                    snd_data 222
                    rcv_data 223
201 [ 8, 4 ] 3      rcv_ack5  49
                    snd_data 223
                    rcv_data 224
202 [10, 3 ] 2      rcv_ack8  78
                    rcv_data 225
203 [ 9, 4 ] 2      rcv_ack7  79
                    snd_data 225
                    rcv_data 226
204 [ 8, 5 ] 2      rcv_ack6  80
                    snd_data 226
                    rcv_data 227
205 [10, 4 ] 1      rcv_ack9 117
                    rcv_data 228
206 [ 9, 5 ] 1      rcv_ack8 118
                    snd_data 228
                    rcv_data 229
207 [ 8, 6 ] 1      rcv_ack7 119
                    snd_data 229
                    rcv_data 230
208 [10, 7 ] 0      rcv_data 231
209 [ 9, 8 ] 0      snd_data 231
                    rcv_data 232
210 [ 8, 0 ] 9      rcv_ack0   0
                    snd_data 233
211 [ 9, 0 ] 9      rcv_ack2   2
                    snd_data 234
                    rcv_data 235
212 [ 8, 1 ] 7      rcv_ack1   3
                    snd_data 235
                    snd_ack  210
213 [10, 0 ] 9      rcv_ack4   7
                    rcv_data 236
214 [ 9, 1 ] 7      rcv_ack3   8
                    snd_data 236
                    rcv_data 237
215 [ 8, 2 ] 6      rcv_ack2   9
                    snd_data 237
                    snd_ack  210
```

```
216 [10, 1 ] 7      rcv_ack5  18
                    rcv_data 238
217 [ 9, 2 ] 5      rcv_ack4  19
                    snd_data 238
                    rcv_data 239
218 [ 8, 3 ] 5      rcv_ack3  20
                    snd_data 239
                    snd_ack  210
219 [10, 2 ] 5      rcv_ack6  36
                    rcv_data 240
220 [ 9, 3 ] 4      rcv_ack5  37
                    snd_data 240
                    rcv_data 241
221 [ 8, 4 ] 4      rcv_ack4  38
                    snd_data 241
                    snd_ack  210
222 [10, 3 ] 3      rcv_ack7  63
                    rcv_data 242
223 [ 9, 4 ] 3      rcv_ack6  64
                    snd_data 242
                    rcv_data 243
224 [ 8, 5 ] 3      rcv_ack5  65
                    snd_data 243
                    snd_ack  210
225 [10, 4 ] 2      rcv_ack8  98
                    rcv_data 244
226 [ 9, 5 ] 2      rcv_ack7  99
                    snd_data 244
                    rcv_data 245
227 [ 8, 6 ] 2      rcv_ack6 100
                    snd_data 245
                    snd_ack  210
228 [10, 5 ] 1      rcv_ack9 141
                    rcv_data 246
229 [ 9, 6 ] 1      rcv_ack8 142
                    snd_data 246
                    rcv_data 247
230 [ 8, 7 ] 1      rcv_ack7 143
                    snd_data 247
                    snd_ack  210
231 [10, 8 ] 0      rcv_data 248
232 [ 9, 9 ] 0      snd_data 248
                    snd_ack  249
233 [ 9, 0 ]10      rcv_ack1   1
                    snd_data 250
                    rcv_data 251
234 [10, 0 ]10      rcv_ack3   4
                    rcv_data 252
235 [ 9, 1 ] 8      rcv_ack2   5
                    snd_data 252
                    rcv_data 253
236 [10, 1 ] 8      rcv_ack4  12
                    rcv_data 254
237 [ 9, 2 ] 6      rcv_ack3  13
                    snd_data 254
                    rcv_data 255
238 [10, 2 ] 6      rcv_ack5  26
                    rcv_data 256
239 [ 9, 3 ] 5      rcv_ack4  27
                    snd_data 256
                    rcv_data 257
240 [10, 3 ] 4      rcv_ack6  48
                    rcv_data 258
241 [ 9, 4 ] 4      rcv_ack5  49
                    snd_data 258
                    rcv_data 259
242 [10, 4 ] 3      rcv_ack7  79
                    rcv_data 260
243 [ 9, 5 ] 3      rcv_ack6  80
                    snd_data 260
                    rcv_data 261
244 [10, 5 ] 2      rcv_ack8 118
                    rcv_data 262
```

```
245 [ 9, 6 ] 2        rcv_ack7 119
                      snd_data 262
                      rcv_data 263
246 [10, 6 ] 1        rcv_ack9 163
                      rcv_data 264
247 [ 9, 7 ] 1        rcv_ack8 164
                      snd_data 264
                      rcv_data 265
248 [10, 9 ] 0        rcv_data 266
249 [ 9, 0 ]11        rcv_ack0   0
                      snd_data 267
250 [10, 0 ]11        rcv_ack2   2
                      rcv_data 268
251 [ 9, 1 ] 9        rcv_ack1   3
                      snd_data 268
                      snd_ack  249
252 [10, 1 ] 9        rcv_ack3   8
                      rcv_data 269
253 [ 9, 2 ] 7        rcv_ack2   9
                      snd_data 269
                      snd_ack  249
254 [10, 2 ] 7        rcv_ack4  19
                      rcv_data 270
255 [ 9, 3 ] 6        rcv_ack3  20
                      snd_data 270
                      snd_ack  249
256 [10, 3 ] 5        rcv_ack5  37
                      rcv_data 271
257 [ 9, 4 ] 5        rcv_ack4  38
                      snd_data 271
                      snd_ack  249
258 [10, 4 ] 4        rcv_ack6  64
                      rcv_data 272
259 [ 9, 5 ] 4        rcv_ack5  65
                      snd_data 272
                      snd_ack  249
260 [10, 5 ] 3        rcv_ack7  99
                      rcv_data 273
261 [ 9, 6 ] 3        rcv_ack6 100
                      snd_data 273
                      snd_ack  249
262 [10, 6 ] 2        rcv_ack8 142
                      rcv_data 274
263 [ 9, 7 ] 2        rcv_ack7 143
                      snd_data 274
                      snd_ack  249
264 [10, 7 ] 1        rcv_ack9 188
                      rcv_data 275
265 [ 9, 8 ] 1        rcv_ack8 189
                      snd_data 275
                      snd_ack  249
266 [10,10 ] 0        snd_ack  276
267 [10, 0 ]12        rcv_ack1   1
                      rcv_data 277
268 [10, 1 ]10        rcv_ack2   5
                      rcv_data 278
269 [10, 2 ] 8        rcv_ack3  13
                      rcv_data 279
270 [10, 3 ] 6        rcv_ack4  27
                      rcv_data 280
271 [10, 4 ] 5        rcv_ack5  49
                      rcv_data 281
272 [10, 5 ] 4        rcv_ack6  80
                      rcv_data 282
273 [10, 6 ] 3        rcv_ack7 119
                      rcv_data 283
274 [10, 7 ] 2        rcv_ack8 164
                      rcv_data 284
275 [10, 8 ] 1        rcv_ack9 209
                      rcv_data 285
276 [10, 0 ]13        rcv_ack0   0
277 [10, 1 ]11        rcv_ack1   3
                      snd_ack  276
```

```
278 [10, 2 ] 9      rcv_ack2   9
                    snd_ack  276
279 [10, 3 ] 7      rcv_ack3  20
                    snd_ack  276
280 [10, 4 ] 6      rcv_ack4  38
                    snd_ack  276
281 [10, 5 ] 5      rcv_ack5  65
                    snd_ack  276
282 [10, 6 ] 4      rcv_ack6 100
                    snd_ack  276
283 [10, 7 ] 3      rcv_ack7 143
                    snd_ack  276
284 [10, 8 ] 2      rcv_ack8 189
                    snd_ack  276
285 [10, 9 ] 1      rcv_ack9 232
                    snd_ack  276

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
---------------------------------------------------------
Number of states generated :286
Number of states analyzed :286
Number of deadlocks : 0

                              UNEXECUTED TRANSITIONS
                                  *****NONE*****
```

APPENDIX C (Token Bus Protocol)

FSM Text File

```
start
number_of_machines 8
machine 1
state 0
trans rcv1 1
trans get_tk1 2
state 1
trans ready1 0
state 2
trans Xmit1 3
trans pass1 0
state 3
trans moreD1 2
trans pass_tk1 0
machine 2
state 0
trans rcv2 1
trans get_tk2 2
state 1
trans ready2 0
state 2
trans Xmit2 3
trans pass2 0
state 3
trans moreD2 2
trans pass_tk2 0
machine 3
state 0
trans rcv3 1
trans get_tk3 2
state 1
trans ready3 0
state 2
trans Xmit3 3
trans pass3 0
state 3
trans moreD3 2
trans pass_tk3 0
machine 4
state 0
trans rcv4 1
trans get_tk4 2
state 1
trans ready4 0
state 2
trans Xmit4 3
trans pass4 0
state 3
trans moreD4 2
trans pass_tk4 0
machine 5
state 0
trans rcv5 1
trans get_tk5 2
state 1
trans ready5 0
state 2
trans Xmit5 3
trans pass5 0
state 3
```

```
trans moreD5 2
trans pass_tk5 0
machine 6
state 0
trans rcv6 1
trans get_tk6 2
state 1
trans ready6 0
state 2
trans Xmit6 3
trans pass6 0
state 3
trans moreD6 2
trans pass_tk6 0
machine 7
state 0
trans rcv7 1
trans get_tk7 2
state 1
trans ready7 0
state 2
trans Xmit7 3
trans pass7 0
state 3
trans moreD7 2
trans pass_tk7 0
machine 8
state 0
trans rcv8 1
trans get_tk8 2
state 1
trans ready8 0
state 2
trans Xmit8 3
trans pass8 0
state 3
trans moreD8 2
trans pass_tk8 0
initial_state 0 0 0 0 0 0 0 0
finish
```

Variable Definitions (No Message in *outbuf* Variables)

```
with TEXT_IO; use TEXT_IO;
package definitions  is
   num_of_machines : constant := 8;
   k : constant := 7; -- number of rows (messages) in output buffer
   type scm_transition_type is (pass1,pass2,pass3, pass4,pass5,pass6,
                                pass7,pass8,get_tk1,get_tk2,
                                get_tk3,get_tk4,get_tk5,get_tk6,
                                get_tk7,get_tk8,Xmit1,Xmit2,Xmit3,
                                Xmit4,Xmit5,Xmit6,Xmit7,Xmit8,moreD1,
                                moreD2,moreD3,moreD4,moreD5,
                                moreD6,moreD7,moreD8,pass_tk4,pass_tk5,
                                pass_tk6,pass_tk7,pass_tk8,
                                pass_tk1,pass_tk2,pass_tk3,
                                rcv1,rcv4,rcv5,rcv6,rcv7,rcv8,
                                rcv2,rcv3,ready1,ready2,ready3,
                                ready4,ready5,ready6,ready7,ready8,unused);

   type dummy_type is range 1..255;
   type t_field_type is (D,T,E);
   package t_field_enum_io is new enumeration_IO(t_field_type);
   use t_field_enum_io;

   type MEDIUM_TYPE is
      record
         t : t_field_type;
         DA : integer range 1..8;
         SA : integer range 1..8;
         data : character;
      end record;

   type input_buffer_type is
      record
        DA : integer range 0..8 :=0;
        SA : integer range 0..8 :=0;
        data : character := 'E';
      end record;

    type output_buffer_type is array (1..k) of MEDIUM_TYPE;


    type machine1_state_type is
       record
         next : integer := 2; --address of downstream neighbor
         i :  integer := 1; -- stations own address
         ctr : integer range 1..(k+1) := 1; -- counter for messages sent
         j : integer range 1..k := 1; -- index for output buffer
         inbuf : input_buffer_type;  -- stores the received messages
         outbuf : output_buffer_type := ((E,2,1,'I'),(E,3,1,'I'),
                                         (E,4,1,'I'),(E,5,1,'I'),
                                         (E,6,1,'I'),(E,7,1,'I'),(E,8,1,'I') );
      end record;

    type machine2_state_type is
       record
         next : integer := 3; --address of downstream neighbor
         i : integer := 2; -- stations own address
         ctr : integer range 1..(k+1):= 1; -- counter for messages sent
         j : integer range 1..k := 1; -- index for output buffer
         inbuf : input_buffer_type;  -- stores the received messages
         outbuf : output_buffer_type := ((E,1,2,'I'),(E,3,2,'I'),
                                         (E,4,2,'I'),(E,5,2,'I'),
                                         (E,6,2,'I'),(E,7,2,'I'),(E,8,2,'I') );
     end record;

     type machine3_state_type is
       record
         next : integer := 4; --address of downstream neighbor
         i : integer := 3; -- stations own address
         ctr : integer range 1..(k+1) := 1; -- counter for messages sent
```

```
      j : integer range 1..k := 1; -- index for output buffer
      inbuf : input_buffer_type;  -- stores the received messages
      outbuf : output_buffer_type := ((E,1,3,'I'),(E,2,3,'I'),
                                      (E,4,3,'I'),(E,5,3,'I'),
                                      (E,6,3,'I'),(E,7,3,'I'),(E,8,3,'I') );
    end record;

    type machine4_state_type is
       record
        next : integer := 5; --address of downstream neighbor
        i : integer := 4; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((E,1,4,'I'),(E,2,4,'I'),(E,3,4,'I'),(E,5,4,'I'),
                                        (E,6,4,'I'),(E,7,4,'I'),(E,8,4,'I') );
       end record;

    type machine5_state_type is
      record
        next : integer := 6; --address of downstream neighbor
        i : integer := 5; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((E,1,5,'I'),(E,2,5,'I'),(E,3,5,'I'),(E,4,5,'I'),
                                        (E,6,5,'I'),(E,7,5,'I'),(E,8,5,'I') );
     end record;

    type machine6_state_type is
       record
        next : integer := 7; --address of downstream neighbor
        i : integer := 6; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((E,1,6,'I'),(E,2,6,'I'),(E,3,6,'I'),(E,4,6,'I'),
                                        (E,5,6,'I'),(E,7,6,'I'),(E,8,6,'I') );
       end record;

    type machine7_state_type is
      record
        next : integer := 8; --address of downstream neighbor
        i : integer := 7; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((E,1,7,'I'),(E,2,7,'I'),(E,3,7,'I'),(E,4,7,'I'),
                                        (E,5,7,'I'),(E,6,7,'I'),(E,8,7,'I') );
      end record;

    type machine8_state_type is
       record
        next : integer := 1; --address of downstream neighbor
        i : integer := 8; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((E,1,8,'I'),(E,2,8,'I'),(E,3,8,'I'),(E,4,8,'I'),
                                        (E,5,8,'I'),(E,6,8,'I'),(E,7,8,'I') );
       end record;

    type global_variable_type is
       record
         MEDIUM : MEDIUM_TYPE :=(T,1,2,'N');
       end record;

end definitions;
```

Variable Definitions(One Message in *outbuf* Variables)

```
with TEXT_IO; use TEXT_IO;
package definitions  is
   num_of_machines : constant := 8;
   k : constant := 7; -- number of rows (messages) in output buffer
   type scm_transition_type is (pass1,pass2,pass3, pass4,pass5,pass6,
                                pass7,pass8,get_tk1,get_tk2,
                                get_tk3,get_tk4,get_tk5,get_tk6,
                                get_tk7,get_tk8,Xmit1,Xmit2,Xmit3,
                                Xmit4,Xmit5,Xmit6,Xmit7,Xmit8,moreD1,
                                moreD2,moreD3,moreD4,moreD5,
                                moreD6,moreD7,moreD8,pass_tk4,pass_tk5,
                                pass_tk6,pass_tk7,pass_tk8,
                                pass_tk1,pass_tk2,pass_tk3,
                                rcv1,rcv4,rcv5,rcv6,rcv7,rcv8,
                                rcv2,rcv3,ready1,ready2,ready3,
                                ready4,ready5,ready6,ready7,ready8,unused);

   type dummy_type is range 1..255;
   type t_field_type is (D,T,E);
   package t_field_enum_io is new enumeration_IO(t_field_type);
   use t_field_enum_io;

   type MEDIUM_TYPE is
      record
         t : t_field_type;
         DA : integer range 1..8;
         SA : integer range 1..8;
         data : character;
      end record;

   type input_buffer_type is
      record
        DA : integer range 0..8 :=0;
        SA : integer range 0..8 :=0;
        data : character := 'E';
      end record;

    type output_buffer_type is array (1..k) of MEDIUM_TYPE;


    type machine1_state_type is
       record
         next : integer := 2; --address of downstream neighbor
         i :  integer := 1; -- stations own address
         ctr : integer range 1..(k+1) := 1; -- counter for messages sent
         j : integer range 1..k := 1; -- index for output buffer
         inbuf : input_buffer_type;  -- stores the received messages
         outbuf : output_buffer_type := ((D,2,1,'I'),(E,3,1,'I'),
                                         (E,4,1,'I'),(E,5,1,'I'),
                                         (E,6,1,'I'),(E,7,1,'I'),(E,8,1,'I') );
      end record;

    type machine2_state_type is
       record
         next : integer := 3; --address of downstream neighbor
         i : integer := 2; -- stations own address
         ctr : integer range 1..(k+1):= 1; -- counter for messages sent
         j : integer range 1..k := 1; -- index for output buffer
         inbuf : input_buffer_type;  -- stores the received messages
         outbuf : output_buffer_type := ((D,1,2,'I'),(E,3,2,'I'),
                                         (E,4,2,'I'),(E,5,2,'I'),
                                         (E,6,2,'I'),(E,7,2,'I'),(E,8,2,'I') );
     end record;

     type machine3_state_type is
       record
         next : integer := 4; --address of downstream neighbor
         i : integer := 3; -- stations own address
         ctr : integer range 1..(k+1) := 1; -- counter for messages sent
```

```
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,3,'I'),(E,2,3,'I'),
                                        (E,4,3,'I'),(E,5,3,'I'),
                                        (E,6,3,'I'),(E,7,3,'I'),(E,8,3,'I') );
    end record;

    type machine4_state_type is
       record
        next : integer := 5; --address of downstream neighbor
        i : integer := 4; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,4,'I'),(E,2,4,'I'),(E,3,4,'I'),(E,5,4,'I'),
                                        (E,6,4,'I'),(E,7,4,'I'),(E,8,4,'I') );
       end record;

    type machine5_state_type is
      record
        next : integer := 6; --address of downstream neighbor
        i : integer := 5; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,5,'I'),(E,2,5,'I'),(E,3,5,'I'),(E,4,5,'I'),
                                        (E,6,5,'I'),(E,7,5,'I'),(E,8,5,'I') );
     end record;

    type machine6_state_type is
       record
        next : integer := 7; --address of downstream neighbor
        i : integer := 6; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,6,'I'),(E,2,6,'I'),(E,3,6,'I'),(E,4,6,'I'),
                                        (E,5,6,'I'),(E,7,6,'I'),(E,8,6,'I') );
       end record;

    type machine7_state_type is
      record
        next : integer := 8; --address of downstream neighbor
        i : integer := 7; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,7,'I'),(E,2,7,'I'),(E,3,7,'I'),(E,4,7,'I'),
                                        (E,5,7,'I'),(E,6,7,'I'),(E,8,7,'I') );
      end record;

    type machine8_state_type is
       record
        next : integer := 1; --address of downstream neighbor
        i : integer := 8; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,8,'I'),(E,2,8,'I'),(E,3,8,'I'),(E,4,8,'I'),
                                        (E,5,8,'I'),(E,6,8,'I'),(E,7,8,'I') );
       end record;

    type global_variable_type is
       record
         MEDIUM : MEDIUM_TYPE :=(T,1,2,'E');
       end record;

end definitions;
```

Variable Definitions

There are seven messages in *outbuf* variable of each machine and each machine sends one message to the other machines in the network.

```
with TEXT_IO; use TEXT_IO;
package definitions  is
   num_of_machines : constant := 8;
   k : constant := 7; -- number of rows (messages) in output buffer
   type scm_transition_type is (pass1,pass2,pass3, pass4,pass5,pass6,
                                pass7,pass8,get_tk1,get_tk2,
                                get_tk3,get_tk4,get_tk5,get_tk6,
                                get_tk7,get_tk8,Xmit1,Xmit2,Xmit3,
                                Xmit4,Xmit5,Xmit6,Xmit7,Xmit8,moreD1,
                                moreD2,moreD3,moreD4,moreD5,
                                moreD6,moreD7,moreD8,pass_tk4,pass_tk5,
                                pass_tk6,pass_tk7,pass_tk8,
                                pass_tk1,pass_tk2,pass_tk3,
                                rcv1,rcv4,rcv5,rcv6,rcv7,rcv8,
                                rcv2,rcv3,ready1,ready2,ready3,
                                ready4,ready5,ready6,ready7,ready8,unused);

   type dummy_type is range 1..255;
   type t_field_type is (D,T,E);
   package t_field_enum_io is new enumeration_IO(t_field_type);
   use t_field_enum_io;

   type MEDIUM_TYPE is
      record
         t : t_field_type;
         DA : integer range 1..8;
         SA : integer range 1..8;
         data : character;
      end record;

   type input_buffer_type is
      record
         DA : integer range 0..8 :=0;
         SA : integer range 0..8 :=0;
         data : character := 'E';
      end record;

    type output_buffer_type is array (1..k) of MEDIUM_TYPE;


    type machine1_state_type is
       record
         next : integer := 2; --address of downstream neighbor
         i :  integer := 1; -- stations own address
         ctr : integer range 1..(k+1) := 1; -- counter for messages sent
         j : integer range 1..k := 1; -- index for output buffer
         inbuf : input_buffer_type;  -- stores the received messages
         outbuf : output_buffer_type := ((D,2,1,'I'),(D,3,1,'I'),
                                         (D,4,1,'I'),(D,5,1,'I'),
                                         (D,6,1,'I'),(D,7,1,'I'),(D,8,1,'I') );
      end record;

    type machine2_state_type is
       record
         next : integer := 3; --address of downstream neighbor
         i : integer := 2; -- stations own address
         ctr : integer range 1..(k+1):= 1; -- counter for messages sent
         j : integer range 1..k := 1; -- index for output buffer
         inbuf : input_buffer_type;  -- stores the received messages
         outbuf : output_buffer_type := ((D,1,2,'I'),(D,3,2,'I'),
                                         (D,4,2,'I'),(D,5,2,'I'),
                                         (D,6,2,'I'),(D,7,2,'I'),(D,8,2,'I') );
     end record;
```

```
   type machine3_state_type is
      record
        next : integer := 4; --address of downstream neighbor
        i : integer := 3; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,3,'I'),(D,2,3,'I'),
                                        (D,4,3,'I'),(D,5,3,'I'),
                                        (D,6,3,'I'),(D,7,3,'I'),(D,8,3,'I') );
    end record;

    type machine4_state_type is
       record
        next : integer := 5; --address of downstream neighbor
        i : integer := 4; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,4,'I'),(D,2,4,'I'),(D,3,4,'I'),(D,5,4,'I'),
                                        (D,6,4,'I'),(D,7,4,'I'),(D,8,4,'I') );
       end record;

    type machine5_state_type is
      record
        next : integer := 6; --address of downstream neighbor
        i : integer := 5; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,5,'I'),(D,2,5,'I'),(D,3,5,'I'),(D,4,5,'I'),
                                        (D,6,5,'I'),(D,7,5,'I'),(D,8,5,'I') );
     end record;

    type machine6_state_type is
       record
        next : integer := 7; --address of downstream neighbor
        i : integer := 6; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,6,'I'),(D,2,6,'I'),(D,3,6,'I'),(D,4,6,'I'),
                                        (D,5,6,'I'),(D,7,6,'I'),(D,8,6,'I') );
       end record;

    type machine7_state_type is
      record
        next : integer := 8; --address of downstream neighbor
        i : integer := 7; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,7,'I'),(D,2,7,'I'),(D,3,7,'I'),(D,4,7,'I'),
                                        (D,5,7,'I'),(D,6,7,'I'),(D,8,7,'I') );
      end record;
    type machine8_state_type is
       record
        next : integer := 1; --address of downstream neighbor
        i : integer := 8; -- stations own address
        ctr : integer range 1..(k+1) := 1; -- counter for messages sent
        j : integer range 1..k := 1; -- index for output buffer
        inbuf : input_buffer_type;  -- stores the received messages
        outbuf : output_buffer_type := ((D,1,8,'I'),(D,2,8,'I'),(D,3,8,'I'),(D,4,8,'I'),
                                        (D,5,8,'I'),(D,6,8,'I'),(D,7,8,'I') );
       end record;
    type global_variable_type is
       record
         MEDIUM : MEDIUM_TYPE :=(T,1,2,'N');
       end record;

end definitions;
```

Predicate-Action Table

```
separate(main)
procedure Analyze_Predicates_Machine1(local : machine1_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is

begin
  case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv1);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)   ) then
        push(w,get_tk1);
      end if;

    when 1 =>
      push(w,ready1);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
        push(w,Xmit1);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass1);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k) ) then
         push(w,moreD1);
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk1);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine1;
---------------------------------------------------------------------------

separate(main)
procedure Analyze_Predicates_Machine2(local : machine2_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is

begin
  case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv2);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)   ) then
        push(w,get_tk2);
      end if;

    when 1 =>
      push(w,ready2);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
        push(w,Xmit2);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass2);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k)   )then
         push(w,moreD2);
```

```
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk2);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine2;
----------------------------------------------------------------------------

separate(main)
procedure Analyze_Predicates_Machine3(local : machine3_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack) is

begin
  case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv3);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)  ) then
        push(w,get_tk3);
      end if;

    when 1 =>
      push(w,ready3);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
        push(w,Xmit3);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass3);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k) )then
         push(w,moreD3);
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk3);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine3;
----------------------------------------------------------------------------

separate(main)
procedure Analyze_Predicates_Machine4(local : machine4_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is


begin
   case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv4);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)  ) then
        push(w,get_tk4);
      end if;

    when 1 =>
      push(w,ready4);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
```

```
        push(w,Xmit4);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass4);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k) )then
         push(w,moreD4);
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk4);
      end if;
    when others =>
      null;
  end case;

end Analyze_Predicates_Machine4;

-----------------------------------------------------------------------

separate(main)
procedure Analyze_Predicates_Machine5(local : machine5_state_type;
                                       global : global_variable_type;
                                       s : natural;
                                       w : in out transition_stack_package.stack)  is


begin
   case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv5);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)  ) then
        push(w,get_tk5);
      end if;

    when 1 =>
      push(w,ready5);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
        push(w,Xmit5);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass5);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k) )then
         push(w,moreD5);
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk5);
      end if;
    when others =>
      null;
  end case;

end Analyze_Predicates_Machine5;

-----------------------------------------------------------------------

separate(main)
procedure Analyze_Predicates_Machine6(local : machine6_state_type;
                                       global : global_variable_type;
                                       s : natural;
                                       w : in out transition_stack_package.stack)  is
```

```
begin
   case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv6);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)  ) then
        push(w,get_tk6);
      end if;

    when 1 =>
      push(w,ready6);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
        push(w,Xmit6);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass6);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k) )then
         push(w,moreD6);
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk6);
      end if;
    when others =>
      null;
  end case;

end Analyze_Predicates_Machine6;

 ---------------------------------------------------------------------------

 separate(main)
procedure Analyze_Predicates_Machine7(local : machine7_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is


begin
   case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv7);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)  ) then
        push(w,get_tk7);
      end if;

    when 1 =>
      push(w,ready7);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
        push(w,Xmit7);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass7);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k) )then
         push(w,moreD7);
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk7);
      end if;
    when others =>
```

```
      null;
  end case;

end Analyze_Predicates_Machine7;

-------------------------------------------------------------------------------

separate(main)
procedure Analyze_Predicates_Machine8(local : machine8_state_type;
                                      global : global_variable_type;
                                      s : natural;
                                      w : in out transition_stack_package.stack)  is


begin
   case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv8);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i)  ) then
        push(w,get_tk8);
      end if;

    when 1 =>
      push(w,ready8);
    when 2 =>
      if  (local.outbuf(local.j).t /= E) then
        push(w,Xmit8);
      end if;
      if ( local.outbuf(local.j).t = E ) then
        push(w,pass8);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
           (local.ctr <= k) )then
         push(w,moreD8);
      end if;
      if ( (global.MEDIUM.t = E ) and ( (local.outbuf(local.j).t = E)
            or (local.ctr = (k+1) ) ) ) then
        push(w, pass_tk8);
      end if;
    when others =>
      null;
  end case;

end Analyze_Predicates_Machine8;


---------------------------------------------------------------------------

separate(main)
procedure Action ( in_system_state : in out Gstate_record_type;
                   in_transition : in out scm_transition_type;
                   out_system_state : in out Gstate_record_type) is

begin
  case in_transition is
    when rcv1 =>
      out_system_state.machine1_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
      out_system_state.machine1_state.inbuf.data
              :=in_system_state.global_variables.MEDIUM.data;
    when rcv2 =>
      out_system_state.machine2_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
      out_system_state.machine2_state.inbuf.data
              :=in_system_state.global_variables.MEDIUM.data;
    when rcv3 =>
      out_system_state.machine3_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
      out_system_state.machine3_state.inbuf.data
```

```
              :=in_system_state.global_variables.MEDIUM.data;
when rcv4 =>
  out_system_state.machine4_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine4_state.inbuf.data
              :=in_system_state.global_variables.MEDIUM.data;
when rcv5 =>
  out_system_state.machine5_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine5_state.inbuf.data
              :=in_system_state.global_variables.MEDIUM.data;
when rcv6 =>
  out_system_state.machine6_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine6_state.inbuf.data
              :=in_system_state.global_variables.MEDIUM.data;
when rcv7 =>
  out_system_state.machine7_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine7_state.inbuf.data
              :=in_system_state.global_variables.MEDIUM.data;
when rcv8=>
  out_system_state.machine8_state.inbuf.SA
              :=in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine8_state.inbuf.data
              :=in_system_state.global_variables.MEDIUM.data;


when ready1 | ready2 | ready3 |ready4|ready5|ready6|ready7|ready8 =>
  out_system_state.global_variables.MEDIUM.t := E ;

when get_tk1 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine1_state.ctr := 1;
when get_tk2 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine2_state.ctr := 1;
when get_tk3 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine3_state.ctr := 1;
when get_tk4 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine4_state.ctr := 1;
when get_tk5 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine5_state.ctr := 1;
when get_tk6 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine6_state.ctr := 1;
when get_tk7 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine7_state.ctr := 1;
when get_tk8 =>
  out_system_state.global_variables.MEDIUM.t := E ;
  out_system_state.machine8_state.ctr := 1;

when pass1 | pass_tk1 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.DA
                    := in_system_state.machine1_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA
                    := in_system_state.machine1_state.i;
when pass2 | pass_tk2 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.DA
                    := in_system_state.machine2_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA
                    := in_system_state.machine2_state.i;
when pass3 | pass_tk3 =>
  out_system_state.global_variables.MEDIUM.t := T;
```

```
    out_system_state.global_variables.MEDIUM.DA
                := in_system_state.machine3_state.next;
    out_system_state.global_variables.MEDIUM.data := 'E';
    out_system_state.global_variables.MEDIUM.SA
                := in_system_state.machine3_state.i;
  when pass4 | pass_tk4 =>
    out_system_state.global_variables.MEDIUM.t := T;
    out_system_state.global_variables.MEDIUM.DA
                := in_system_state.machine4_state.next;
    out_system_state.global_variables.MEDIUM.data := 'E';
    out_system_state.global_variables.MEDIUM.SA
                := in_system_state.machine4_state.i;
  when pass5 | pass_tk5 =>
    out_system_state.global_variables.MEDIUM.t := T;
    out_system_state.global_variables.MEDIUM.DA
                := in_system_state.machine5_state.next;
    out_system_state.global_variables.MEDIUM.data := 'E';
    out_system_state.global_variables.MEDIUM.SA
                := in_system_state.machine5_state.i;
  when pass6 | pass_tk6 =>
    out_system_state.global_variables.MEDIUM.t := T;
    out_system_state.global_variables.MEDIUM.DA
                := in_system_state.machine6_state.next;
    out_system_state.global_variables.MEDIUM.data := 'E';
    out_system_state.global_variables.MEDIUM.SA
                := in_system_state.machine6_state.i;
  when pass7 | pass_tk7 =>
    out_system_state.global_variables.MEDIUM.t := T;
    out_system_state.global_variables.MEDIUM.DA
                := in_system_state.machine7_state.next;
    out_system_state.global_variables.MEDIUM.data := 'E';
    out_system_state.global_variables.MEDIUM.SA
                := in_system_state.machine7_state.i;
  when pass8 | pass_tk8 =>
    out_system_state.global_variables.MEDIUM.t := T;
    out_system_state.global_variables.MEDIUM.DA
                := in_system_state.machine8_state.next;
    out_system_state.global_variables.MEDIUM.data := 'E';
    out_system_state.global_variables.MEDIUM.SA
                := in_system_state.machine8_state.i;


  when Xmit1 =>
     out_system_state.global_variables.MEDIUM
      := in_system_state.machine1_state.outbuf(in_system_state.machine1_state.j);
     out_system_state.machine1_state.outbuf(in_system_state.machine1_state.j).t := E;
     out_system_state.machine1_state.ctr
               := (in_system_state.machine1_state.ctr mod 8) + 1;
     out_system_state.machine1_state.j
               := (in_system_state.machine1_state.j mod 7) + 1;
  when Xmit2 =>
     out_system_state.global_variables.MEDIUM
      := in_system_state.machine2_state.outbuf(in_system_state.machine2_state.j);
      out_system_state.machine2_state.outbuf(in_system_state.machine2_state.j).t := E;
     out_system_state.machine2_state.ctr
               := (in_system_state.machine2_state.ctr mod 8) + 1;
     out_system_state.machine2_state.j
               := (in_system_state.machine2_state.j mod 7) + 1;
  when Xmit3 =>
     out_system_state.global_variables.MEDIUM
      := in_system_state.machine3_state.outbuf(in_system_state.machine3_state.j);
      out_system_state.machine3_state.outbuf(in_system_state.machine3_state.j).t := E;
     out_system_state.machine3_state.ctr
               := (in_system_state.machine3_state.ctr mod 8) + 1;
     out_system_state.machine3_state.j
               := (in_system_state.machine3_state.j mod 7) + 1;
   when Xmit4 =>
     out_system_state.global_variables.MEDIUM
      := in_system_state.machine4_state.outbuf(in_system_state.machine4_state.j);
     out_system_state.machine4_state.outbuf(in_system_state.machine4_state.j).t := E;
     out_system_state.machine4_state.ctr
               := (in_system_state.machine4_state.ctr mod 8) + 1;
```

```
      out_system_state.machine4_state.j
                    := (in_system_state.machine4_state.j mod 7) + 1;
   when Xmit5 =>
      out_system_state.global_variables.MEDIUM
        := in_system_state.machine5_state.outbuf(in_system_state.machine5_state.j);
       out_system_state.machine5_state.outbuf(in_system_state.machine5_state.j).t := E;
      out_system_state.machine5_state.ctr
                    := (in_system_state.machine5_state.ctr mod 8) + 1;
      out_system_state.machine5_state.j
                    := (in_system_state.machine5_state.j mod 7) + 1;
   when Xmit6 =>
      out_system_state.global_variables.MEDIUM
        := in_system_state.machine6_state.outbuf(in_system_state.machine6_state.j);
       out_system_state.machine6_state.outbuf(in_system_state.machine6_state.j).t := E;
      out_system_state.machine6_state.ctr
                    := (in_system_state.machine6_state.ctr mod 8) + 1;
      out_system_state.machine6_state.j
                    := (in_system_state.machine6_state.j mod 7) + 1;
  when Xmit7 =>
      out_system_state.global_variables.MEDIUM
       := in_system_state.machine7_state.outbuf(in_system_state.machine7_state.j);
       out_system_state.machine7_state.outbuf(in_system_state.machine7_state.j).t := E;
      out_system_state.machine7_state.ctr
                    := (in_system_state.machine7_state.ctr mod 8) + 1;
      out_system_state.machine7_state.j
                    := (in_system_state.machine7_state.j mod 7) + 1;
   when Xmit8 =>
      out_system_state.global_variables.MEDIUM
       := in_system_state.machine8_state.outbuf(in_system_state.machine8_state.j);
       out_system_state.machine8_state.outbuf(in_system_state.machine8_state.j).t := E;
      out_system_state.machine8_state.ctr
                     := (in_system_state.machine8_state.ctr mod 8) + 1;
      out_system_state.machine8_state.j
                     := (in_system_state.machine8_state.j mod 7) + 1;
   when moreD1 | moreD2 | moreD3|moreD4|moreD5|moreD6|moreD7|moreD8 =>
     null;
   when others =>
     put("Error in action procedure");
  end case;
end Action;
```

Output Format

```
separate(main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
  if print_header then
    new_line(2);
    set_col(7);
    put_line("m1,m2,m3,m4,m5,m6,m7,m8,MEDIUM.t,MEDIUM.DA,MEDIUM.SA,MEDIUM.data");
    print_header := false;
  else
    put("  ["& integer'image(tuple.machine_state(1)) );
    put(" , ");
    put( integer'image(tuple.machine_state(2)) );
    put(" , ");
    put( integer'image(tuple.machine_state(3)) );
    put(" , ");
    put( integer'image(tuple.machine_state(4)) );
    put(" , ");
    put( integer'image(tuple.machine_state(5)) );
    put(" , ");
    put( integer'image(tuple.machine_state(6)) );
    put(" , ");
    put( integer'image(tuple.machine_state(7)) );
    put(" , ");
    put( integer'image(tuple.machine_state(8)) );
    put(" , ");
    t_field_enum_io.put(tuple.global_variables.MEDIUM.t, set => upper_case);
    put(" , ");
    put(tuple.global_variables.MEDIUM.DA, width => 1);
    put(" , ");
    put(tuple.global_variables.MEDIUM.SA, width => 1);
    put(" , ");
    put(tuple.global_variables.MEDIUM.data);
    put(" ]");
  end if;
end output_Gtuple;
```

Program Output (No Message in *outbuf* Variable)

```
REACHABILITY ANALYSIS of :tb8.scm
           SPECIFICATION

 ------------------------------
| Machine 1 State Transitions |
 ------------------------------
| From  |  To  |  Transition  |
 ------------------------------
|   0   |   1  |  rcv1        |
|   0   |   2  |  get_tk1     |
|   1   |   0  |  ready1      |
|   2   |   3  |  xmit1       |
|   2   |   0  |  pass1       |
|   3   |   2  |  mored1      |
|   3   |   0  |  pass_tk1    |
 ------------------------------
 ------------------------------
| Machine 2 State Transitions |
 ------------------------------
| From  |  To  |  Transition  |
 ------------------------------
|   0   |   1  |  rcv2        |
|   0   |   2  |  get_tk2     |
|   1   |   0  |  ready2      |
|   2   |   3  |  xmit2       |
|   2   |   0  |  pass2       |
|   3   |   2  |  mored2      |
|   3   |   0  |  pass_tk2    |
 ------------------------------
 ------------------------------
| Machine 3 State Transitions |
 ------------------------------
| From  |  To  |  Transition  |
 ------------------------------
|   0   |   1  |  rcv3        |
|   0   |   2  |  get_tk3     |
|   1   |   0  |  ready3      |
|   2   |   3  |  xmit3       |
|   2   |   0  |  pass3       |
|   3   |   2  |  mored3      |
|   3   |   0  |  pass_tk3    |
 ------------------------------
 ------------------------------
| Machine 4 State Transitions |
 ------------------------------
| From  |  To  |  Transition  |
 ------------------------------
|   0   |   1  |  rcv4        |
|   0   |   2  |  get_tk4     |
|   1   |   0  |  ready4      |
|   2   |   3  |  xmit4       |
|   2   |   0  |  pass4       |
|   3   |   2  |  mored4      |
|   3   |   0  |  pass_tk4    |
 ------------------------------
 ------------------------------
| Machine 5 State Transitions |
 ------------------------------
| From  |  To  |  Transition  |
 ------------------------------
|   0   |   1  |  rcv5        |
|   0   |   2  |  get_tk5     |
|   1   |   0  |  ready5      |
|   2   |   3  |  xmit5       |
|   2   |   0  |  pass5       |
|   3   |   2  |  mored5      |
|   3   |   0  |  pass_tk5    |
 ------------------------------
```

Machine 6 State Transitions		
From	To	Transition
0	1	rcv6
0	2	get_tk6
1	0	ready6
2	3	xmit6
2	0	pass6
3	2	mored6
3	0	pass_tk6

Machine 7 State Transitions		
From	To	Transition
0	1	rcv7
0	2	get_tk7
1	0	ready7
2	3	xmit7
2	0	pass7
3	2	mored7
3	0	pass_tk7

Machine 8 State Transitions		
From	To	Transition
0	1	rcv8
0	2	get_tk8
1	0	ready8
2	3	xmit8
2	0	pass8
3	2	mored8
3	0	pass_tk8

SYSTEM REACHABILITY GRAPH

```
 0 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 0    get_tk1    1

 1 [ 2, 0, 0, 0, 0, 0, 0, 0 ] 0    pass1      2

 2 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 1    get_tk2    3

 3 [ 0, 2, 0, 0, 0, 0, 0, 0 ] 0    pass2      4

 4 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 2    get_tk3    5

 5 [ 0, 0, 2, 0, 0, 0, 0, 0 ] 0    pass3      6

 6 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 3    get_tk4    7

 7 [ 0, 0, 0, 2, 0, 0, 0, 0 ] 0    pass4      8

 8 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 4    get_tk5    9

 9 [ 0, 0, 0, 0, 2, 0, 0, 0 ] 0    pass5     10

10 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 5    get_tk6   11

11 [ 0, 0, 0, 0, 0, 2, 0, 0 ] 0    pass6     12

12 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 6    get_tk7   13

13 [ 0, 0, 0, 0, 0, 0, 2, 0 ] 0    pass7     14
```

```
14 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 7   get_tk8   15

15 [ 0, 0, 0, 0, 0, 0, 0, 2 ] 0   pass8      0
```

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)

```
Number of states generated :16
Number of states analyzed :16
Number of deadlocks : 0
```

UNEXECUTED TRANSITIONS

Machine 1 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv1
1	0	ready1
2	3	xmit1
3	2	mored1
3	0	pass_tk1

Machine 2 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv2
1	0	ready2
2	3	xmit2
3	2	mored2
3	0	pass_tk2

Machine 3 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv3
1	0	ready3
2	3	xmit3
3	2	mored3
3	0	pass_tk3

Machine 4 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv4
1	0	ready4
2	3	xmit4
3	2	mored4
3	0	pass_tk4

Machine 5 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv5
1	0	ready5
2	3	xmit5
3	2	mored5
3	0	pass_tk5

Machine 6 Unexecuted Transitions		
From	**To**	**Unexecuted Transition**
0	1	rcv6
1	0	ready6
2	3	xmit6
3	2	mored6
3	0	pass_tk6

Machine 7 Unexecuted Transitions		
From	**To**	**Unexecuted Transition**
0	1	rcv7
1	0	ready7
2	3	xmit7
3	2	mored7
3	0	pass_tk7

Machine 8 Unexecuted Transitions		
From	**To**	**Unexecuted Transition**
0	1	rcv8
1	0	ready8
2	3	xmit8
3	2	mored8
3	0	pass_tk8

Program Output (One Message in *outbuf* Variable)

```
          SYSTEM REACHABILITY GRAPH
 0 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 0  get_tk1    1

 1 [ 2, 0, 0, 0, 0, 0, 0, 0 ] 0  xmit1      2

 2 [ 3, 0, 0, 0, 0, 0, 0, 0 ] 0  rcv2       3

 3 [ 3, 1, 0, 0, 0, 0, 0, 0 ] 0  ready2     4

 4 [ 3, 0, 0, 0, 0, 0, 0, 0 ] 1  pass_tk1   5

 5 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 1  get_tk2    6

 6 [ 0, 2, 0, 0, 0, 0, 0, 0 ] 0  xmit2      7

 7 [ 0, 3, 0, 0, 0, 0, 0, 0 ] 0  rcv1       8

 8 [ 1, 3, 0, 0, 0, 0, 0, 0 ] 0  ready1     9

 9 [ 0, 3, 0, 0, 0, 0, 0, 0 ] 1  pass_tk2  10

10 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 2  get_tk3   11

11 [ 0, 0, 2, 0, 0, 0, 0, 0 ] 0  xmit3     12

12 [ 0, 0, 3, 0, 0, 0, 0, 0 ] 0  rcv1      13

13 [ 1, 0, 3, 0, 0, 0, 0, 0 ] 0  ready1    14

14 [ 0, 0, 3, 0, 0, 0, 0, 0 ] 1  pass_tk3  15

15 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 3  get_tk4   16

16 [ 0, 0, 0, 2, 0, 0, 0, 0 ] 0  xmit4     17

17 [ 0, 0, 0, 3, 0, 0, 0, 0 ] 0  rcv1      18

18 [ 1, 0, 0, 3, 0, 0, 0, 0 ] 0  ready1    19

19 [ 0, 0, 0, 3, 0, 0, 0, 0 ] 1  pass_tk4  20

20 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 4  get_tk5   21

21 [ 0, 0, 0, 0, 2, 0, 0, 0 ] 0  xmit5     22

22 [ 0, 0, 0, 0, 3, 0, 0, 0 ] 0  rcv1      23

23 [ 1, 0, 0, 0, 3, 0, 0, 0 ] 0  ready1    24

24 [ 0, 0, 0, 0, 3, 0, 0, 0 ] 1  pass_tk5  25

25 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 5  get_tk6   26

26 [ 0, 0, 0, 0, 0, 2, 0, 0 ] 0  xmit6     27

27 [ 0, 0, 0, 0, 0, 3, 0, 0 ] 0  rcv1      28

28 [ 1, 0, 0, 0, 0, 3, 0, 0 ] 0  ready1    29

29 [ 0, 0, 0, 0, 0, 3, 0, 0 ] 1  pass_tk6  30

30 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 6  get_tk7   31

31 [ 0, 0, 0, 0, 0, 0, 2, 0 ] 0  xmit7     32

32 [ 0, 0, 0, 0, 0, 0, 3, 0 ] 0  rcv1      33

33 [ 1, 0, 0, 0, 0, 0, 3, 0 ] 0  ready1    34
```

```
34 [ 0, 0, 0, 0, 0, 0, 3, 0 ] 1  pass_tk7  35
35 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 7  get_tk8   36
36 [ 0, 0, 0, 0, 0, 0, 0, 2 ] 0  xmit8     37
37 [ 0, 0, 0, 0, 0, 0, 0, 3 ] 0  rcv1      38
38 [ 1, 0, 0, 0, 0, 0, 0, 3 ] 0  ready1    39
39 [ 0, 0, 0, 0, 0, 0, 0, 3 ] 1  pass_tk8   0
```

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)

Number of states generated :40
Number of states analyzed :40
Number of deadlocks : 0

UNEXECUTED TRANSITIONS

Machine 1 Unexecuted Transitions		
From	To	Unexecuted Transition
2	0	pass1
3	2	mored1

Machine 2 Unexecuted Transitions		
From	To	Unexecuted Transition
2	0	pass2
3	2	mored2

Machine 3 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv3
1	0	ready3
2	0	pass3
3	2	mored3

Machine 4 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv4
1	0	ready4
2	0	pass4
3	2	mored4

Machine 5 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv5
1	0	ready5
2	0	pass5
3	2	mored5

Machine 6 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv6
1	0	ready6
2	0	pass6
3	2	mored6

Machine 7 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv7
1	0	ready7
2	0	pass7
3	2	mored7

Machine 8 Unexecuted Transitions		
From	To	Unexecuted Transition
0	1	rcv8
1	0	ready8
2	0	pass8
3	2	mored8

Program Output (More Than One Message in *outbuf* Variable)

```
         SYSTEM REACHABILITY GRAPH
0 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 0  get_tk1    1

1 [ 2, 0, 0, 0, 0, 0, 0, 0 ] 0  xmit1      2

2 [ 3, 0, 0, 0, 0, 0, 0, 0 ] 0  rcv2       3

3 [ 3, 1, 0, 0, 0, 0, 0, 0 ] 0  ready2     4

4 [ 3, 0, 0, 0, 0, 0, 0, 0 ] 1  mored1     1
```

```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
------------------------------------------------------
Number of states generated :5
Number of states analyzed :5
Number of deadlocks : 0
```

UNEXECUTED TRANSITIONS

Machine 1 Unexecuted Transitions

From	To	Unexecuted Transition
0	1	rcv1
1	0	ready1
2	0	pass1
3	0	pass_tk1

Machine 2 Unexecuted Transitions

From	To	Unexecuted Transition
0	2	get_tk2
2	3	xmit2
2	0	pass2
3	2	mored2
3	0	pass_tk2

Machine 3 Unexecuted Transitions

From	To	Unexecuted Transition
0	1	rcv3
0	2	get_tk3
1	0	ready3
2	3	xmit3
2	0	pass3
3	2	mored3
3	0	pass_tk3

Machine 4 Unexecuted Transitions

From	To	Unexecuted Transition
0	1	rcv4
0	2	get_tk4
1	0	ready4
2	3	xmit4
2	0	pass4
3	2	mored4
3	0	pass_tk4

Machine 5 Unexecuted Transitions

From	To	Unexecuted Transition
0	1	rcv5
0	2	get_tk5
1	0	ready5
2	3	xmit5
2	0	pass5
3	2	mored5
3	0	pass_tk5

Machine 6 Unexecuted Transitions

From	To	Unexecuted Transition
0	1	rcv6
0	2	get_tk6
1	0	ready6
2	3	xmit6
2	0	pass6
3	2	mored6
3	0	pass_tk6

Machine 7 Unexecuted Transitions

From	To	Unexecuted Transition
0	1	rcv7
0	2	get_tk7
1	0	ready7
2	3	xmit7
2	0	pass7
3	2	mored7
3	0	pass_tk7

Machine 8 Unexecuted Transitions

From	To	Unexecuted Transition
0	1	rcv8
0	2	get_tk8
1	0	ready8
2	3	xmit8
2	0	pass8
3	2	mored8
3	0	pass_tk8

Program Output (Global Reachability Analysis)

There are seven messages in *outbuf* variable of each machine.

```
REACHABILITY ANALYSIS of :tb8.scm
         SPECIFICATION
------------------------------
| Machine 1 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |  1   |  rcv1        |
|   0   |  2   |  get_tk1     |
|   1   |  0   |  ready1      |
|   2   |  3   |  xmit1       |
|   2   |  0   |  pass1       |
|   3   |  2   |  mored1      |
|   3   |  0   |  pass_tk1    |
-------------------------------
------------------------------
| Machine 2 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |  1   |  rcv2        |
|   0   |  2   |  get_tk2     |
|   1   |  0   |  ready2      |
|   2   |  3   |  xmit2       |
|   2   |  0   |  pass2       |
|   3   |  2   |  mored2      |
|   3   |  0   |  pass_tk2    |
-------------------------------
------------------------------
| Machine 3 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |  1   |  rcv3        |
|   0   |  2   |  get_tk3     |
|   1   |  0   |  ready3      |
|   2   |  3   |  xmit3       |
|   2   |  0   |  pass3       |
|   3   |  2   |  mored3      |
|   3   |  0   |  pass_tk3    |
-------------------------------
------------------------------
| Machine 4 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |  1   |  rcv4        |
|   0   |  2   |  get_tk4     |
|   1   |  0   |  ready4      |
|   2   |  3   |  xmit4       |
|   2   |  0   |  pass4       |
|   3   |  2   |  mored4      |
|   3   |  0   |  pass_tk4    |
-------------------------------
------------------------------
| Machine 5 State Transitions |
------------------------------
| From  |  To  |  Transition  |
------------------------------
|   0   |  1   |  rcv5        |
|   0   |  2   |  get_tk5     |
|   1   |  0   |  ready5      |
|   2   |  3   |  xmit5       |
|   2   |  0   |  pass5       |
|   3   |  2   |  mored5      |
|   3   |  0   |  pass_tk5    |
-------------------------------
```

Machine 6 State Transitions		
From	To	Transition
0	1	rcv6
0	2	get_tk6
1	0	ready6
2	3	xmit6
2	0	pass6
3	2	mored6
3	0	pass_tk6

Machine 7 State Transitions		
From	To	Transition
0	1	rcv7
0	2	get_tk7
1	0	ready7
2	3	xmit7
2	0	pass7
3	2	mored7
3	0	pass_tk7

Machine 8 State Transitions		
From	To	Transition
0	1	rcv8
0	2	get_tk8
1	0	ready8
2	3	xmit8
2	0	pass8
3	2	mored8
3	0	pass_tk8

REACHABILITY GRAPH

```
 [m1,m2,m3,m4,m5,m6,m7,m8,MEDIUM.t,MEDIUM.DA,MEDIUM.SA,MEDIUM.data]

 0  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 1 , 2 , E ] get_tk1    1
 1  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 1 , 2 , E ] xmit1      2
 2  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , D , 2 , 1 , I ] rcv2       3
 3  [ 3 , 1 , 0 , 0 , 0 , 0 , 0 , 0 , D , 2 , 1 , I ] ready2     4
 4  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 2 , 1 , I ] mored1     5
 5  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 2 , 1 , I ] xmit1      6
 6  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , D , 3 , 1 , I ] rcv3       7
 7  [ 3 , 0 , 1 , 0 , 0 , 0 , 0 , 0 , D , 3 , 1 , I ] ready3     8
 8  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 3 , 1 , I ] mored1     9
 9  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 3 , 1 , I ] xmit1      10
10  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , D , 4 , 1 , I ] rcv4       11
11  [ 3 , 0 , 0 , 1 , 0 , 0 , 0 , 0 , D , 4 , 1 , I ] ready4     12
12  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 4 , 1 , I ] mored1     13
13  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 4 , 1 , I ] xmit1      14
14  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , D , 5 , 1 , I ] rcv5       15
15  [ 3 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , D , 5 , 1 , I ] ready5     16
16  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 5 , 1 , I ] mored1     17
17  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 5 , 1 , I ] xmit1      18
18  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , D , 6 , 1 , I ] rcv6       19
19  [ 3 , 0 , 0 , 0 , 0 , 1 , 0 , 0 , D , 6 , 1 , I ] ready6     20
20  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 6 , 1 , I ] mored1     21
21  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 6 , 1 , I ] xmit1      22
22  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , D , 7 , 1 , I ] rcv7       23
23  [ 3 , 0 , 0 , 0 , 0 , 0 , 1 , 0 , D , 7 , 1 , I ] ready7     24
24  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 7 , 1 , I ] mored1     25
25  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 7 , 1 , I ] xmit1      26
26  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , D , 8 , 1 , I ] rcv8       27
27  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 1 , D , 8 , 1 , I ] ready8     28
28  [ 3 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 8 , 1 , I ] pass_tk1   29
29  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 2 , 1 , E ] get_tk2    30
```

```
 30  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 2 , 1 , E ] xmit2     31
 31  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 1 , 2 , I ] rcv1      32
 32  [ 1 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 1 , 2 , I ] ready1    33
 33  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , E , 1 , 2 , I ] mored2    34
 34  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 1 , 2 , I ] xmit2     35
 35  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 3 , 2 , I ] rcv3      36
 36  [ 0 , 3 , 1 , 0 , 0 , 0 , 0 , 0 , D , 3 , 2 , I ] ready3    37
 37  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , E , 3 , 2 , I ] mored2    38
 38  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 3 , 2 , I ] xmit2     39
 39  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 4 , 2 , I ] rcv4      40
 40  [ 0 , 3 , 0 , 1 , 0 , 0 , 0 , 0 , D , 4 , 2 , I ] ready4    41
 41  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , E , 4 , 2 , I ] mored2    42
 42  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 4 , 2 , I ] xmit2     43
 43  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 5 , 2 , I ] rcv5      44
 44  [ 0 , 3 , 0 , 0 , 1 , 0 , 0 , 0 , D , 5 , 2 , I ] ready5    45
 45  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , E , 5 , 2 , I ] mored2    46
 46  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 5 , 2 , I ] xmit2     47
 47  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 6 , 2 , I ] rcv6      48
 48  [ 0 , 3 , 0 , 0 , 0 , 1 , 0 , 0 , D , 6 , 2 , I ] ready6    49
 49  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , E , 6 , 2 , I ] mored2    50
 50  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 6 , 2 , I ] xmit2     51
 51  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 7 , 2 , I ] rcv7      52
 52  [ 0 , 3 , 0 , 0 , 0 , 0 , 1 , 0 , D , 7 , 2 , I ] ready7    53
 53  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , E , 7 , 2 , I ] mored2    54
 54  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 7 , 2 , I ] xmit2     55
 55  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , D , 8 , 2 , I ] rcv8      56
 56  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 1 , D , 8 , 2 , I ] ready8    57
 57  [ 0 , 3 , 0 , 0 , 0 , 0 , 0 , 0 , E , 8 , 2 , I ] pass_tk2  58
 58  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 3 , 2 , E ] get_tk3   59
 59  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 3 , 2 , E ] xmit3     60
 60  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 1 , 3 , I ] rcv1      61
 61  [ 1 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 1 , 3 , I ] ready1    62
 62  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , E , 1 , 3 , I ] mored3    63
 63  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 1 , 3 , I ] xmit3     64
 64  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 2 , 3 , I ] rcv2      65
 65  [ 0 , 1 , 3 , 0 , 0 , 0 , 0 , 0 , D , 2 , 3 , I ] ready2    66
 66  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , E , 2 , 3 , I ] mored3    67
 67  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 2 , 3 , I ] xmit3     68
 68  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 4 , 3 , I ] rcv4      69
 69  [ 0 , 0 , 3 , 1 , 0 , 0 , 0 , 0 , D , 4 , 3 , I ] ready4    70
 70  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , E , 4 , 3 , I ] mored3    71
 71  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 4 , 3 , I ] xmit3     72
 72  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 5 , 3 , I ] rcv5      73
 73  [ 0 , 0 , 3 , 0 , 1 , 0 , 0 , 0 , D , 5 , 3 , I ] ready5    74
 74  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , E , 5 , 3 , I ] mored3    75
 75  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 5 , 3 , I ] xmit3     76
 76  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 6 , 3 , I ] rcv6      77
 77  [ 0 , 0 , 3 , 0 , 0 , 1 , 0 , 0 , D , 6 , 3 , I ] ready6    78
 78  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , E , 6 , 3 , I ] mored3    79
 79  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 6 , 3 , I ] xmit3     80
 80  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 7 , 3 , I ] rcv7      81
 81  [ 0 , 0 , 3 , 0 , 0 , 0 , 1 , 0 , D , 7 , 3 , I ] ready7    82
 82  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , E , 7 , 3 , I ] mored3    83
 83  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 7 , 3 , I ] xmit3     84
 84  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , D , 8 , 3 , I ] rcv8      85
 85  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 1 , D , 8 , 3 , I ] ready8    86
 86  [ 0 , 0 , 3 , 0 , 0 , 0 , 0 , 0 , E , 8 , 3 , I ] pass_tk3  87
 87  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 4 , 3 , E ] get_tk4   88
 88  [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 4 , 3 , E ] xmit4     89
 89  [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 1 , 4 , I ] rcv1      90
 90  [ 1 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 1 , 4 , I ] ready1    91
 91  [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , E , 1 , 4 , I ] mored4    92
 92  [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 1 , 4 , I ] xmit4     93
 93  [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 2 , 4 , I ] rcv2      94
 94  [ 0 , 1 , 0 , 3 , 0 , 0 , 0 , 0 , D , 2 , 4 , I ] ready2    95
 95  [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , E , 2 , 4 , I ] mored4    96
 96  [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 2 , 4 , I ] xmit4     97
 97  [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 3 , 4 , I ] rcv3      98
 98  [ 0 , 0 , 1 , 3 , 0 , 0 , 0 , 0 , D , 3 , 4 , I ] ready3    99
 99  [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , E , 3 , 4 , I ] mored4   100
100  [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 3 , 4 , I ] xmit4    101
101  [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 5 , 4 , I ] rcv5     102
```

```
102 [ 0 , 0 , 0 , 3 , 1 , 0 , 0 , 0 , D , 5 , 4 , I ] ready5   103
103 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , E , 5 , 4 , I ] mored4   104
104 [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 5 , 4 , I ] xmit4    105
105 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 6 , 4 , I ] rcv6     106
106 [ 0 , 0 , 0 , 3 , 0 , 1 , 0 , 0 , D , 6 , 4 , I ] ready6   107
107 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , E , 6 , 4 , I ] mored4   108
108 [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 6 , 4 , I ] xmit4    109
109 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 7 , 4 , I ] rcv7     110
110 [ 0 , 0 , 0 , 3 , 0 , 0 , 1 , 0 , D , 7 , 4 , I ] ready7   111
111 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , E , 7 , 4 , I ] mored4   112
112 [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 7 , 4 , I ] xmit4    113
113 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , D , 8 , 4 , I ] rcv8     114
114 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 1 , D , 8 , 4 , I ] ready8   115
115 [ 0 , 0 , 0 , 3 , 0 , 0 , 0 , 0 , E , 8 , 4 , I ] pass_tk4 116
116 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 5 , 4 , E ] get_tk5  117
117 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 5 , 4 , E ] xmit5    118
118 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 1 , 5 , I ] rcv1     119
119 [ 1 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 1 , 5 , I ] ready1   120
120 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , E , 1 , 5 , I ] mored5   121
121 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 1 , 5 , I ] xmit5    122
122 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 2 , 5 , I ] rcv2     123
123 [ 0 , 1 , 0 , 0 , 3 , 0 , 0 , 0 , D , 2 , 5 , I ] ready2   124
124 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , E , 2 , 5 , I ] mored5   125
125 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 2 , 5 , I ] xmit5    126
126 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 3 , 5 , I ] rcv3     127
127 [ 0 , 0 , 1 , 0 , 3 , 0 , 0 , 0 , D , 3 , 5 , I ] ready3   128
128 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , E , 3 , 5 , I ] mored5   129
129 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 3 , 5 , I ] xmit5    130
130 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 4 , 5 , I ] rcv4     131
131 [ 0 , 0 , 0 , 1 , 3 , 0 , 0 , 0 , D , 4 , 5 , I ] ready4   132
132 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , E , 4 , 5 , I ] mored5   133
133 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 4 , 5 , I ] xmit5    134
134 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 6 , 5 , I ] rcv6     135
135 [ 0 , 0 , 0 , 0 , 3 , 1 , 0 , 0 , D , 6 , 5 , I ] ready6   136
136 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , E , 6 , 5 , I ] mored5   137
137 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 6 , 5 , I ] xmit5    138
138 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 7 , 5 , I ] rcv7     139
139 [ 0 , 0 , 0 , 0 , 3 , 0 , 1 , 0 , D , 7 , 5 , I ] ready7   140
140 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , E , 7 , 5 , I ] mored5   141
141 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 7 , 5 , I ] xmit5    142
142 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , D , 8 , 5 , I ] rcv8     143
143 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 1 , D , 8 , 5 , I ] ready8   144
144 [ 0 , 0 , 0 , 0 , 3 , 0 , 0 , 0 , E , 8 , 5 , I ] pass_tk5 145
145 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 6 , 5 , E ] get_tk6  146
146 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 6 , 5 , E ] xmit6    147
147 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 1 , 6 , I ] rcv1     148
148 [ 1 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 1 , 6 , I ] ready1   149
149 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , E , 1 , 6 , I ] mored6   150
150 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 1 , 6 , I ] xmit6    151
151 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 2 , 6 , I ] rcv2     152
152 [ 0 , 1 , 0 , 0 , 0 , 3 , 0 , 0 , D , 2 , 6 , I ] ready2   153
153 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , E , 2 , 6 , I ] mored6   154
154 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 2 , 6 , I ] xmit6    155
155 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 3 , 6 , I ] rcv3     156
156 [ 0 , 0 , 1 , 0 , 0 , 3 , 0 , 0 , D , 3 , 6 , I ] ready3   157
157 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , E , 3 , 6 , I ] mored6   158
158 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 3 , 6 , I ] xmit6    159
159 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 4 , 6 , I ] rcv4     160
160 [ 0 , 0 , 0 , 1 , 0 , 3 , 0 , 0 , D , 4 , 6 , I ] ready4   161
161 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , E , 4 , 6 , I ] mored6   162
162 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 4 , 6 , I ] xmit6    163
163 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 5 , 6 , I ] rcv5     164
164 [ 0 , 0 , 0 , 0 , 1 , 3 , 0 , 0 , D , 5 , 6 , I ] ready5   165
165 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , E , 5 , 6 , I ] mored6   166
166 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 5 , 6 , I ] xmit6    167
167 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 7 , 6 , I ] rcv7     168
168 [ 0 , 0 , 0 , 0 , 0 , 3 , 1 , 0 , D , 7 , 6 , I ] ready7   169
169 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , E , 7 , 6 , I ] mored6   170
170 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 7 , 6 , I ] xmit6    171
171 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , D , 8 , 6 , I ] rcv8     172
172 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 1 , D , 8 , 6 , I ] ready8   173
173 [ 0 , 0 , 0 , 0 , 0 , 3 , 0 , 0 , E , 8 , 6 , I ] pass_tk6 174
```

```
174 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 7 , 6 , E ] get_tk7  175
175 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 7 , 6 , E ] xmit7    176
176 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 1 , 7 , I ] rcv1     177
177 [ 1 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 1 , 7 , I ] ready1   178
178 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , E , 1 , 7 , I ] mored7   179
179 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 1 , 7 , I ] xmit7    180
180 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 2 , 7 , I ] rcv2     181
181 [ 0 , 1 , 0 , 0 , 0 , 0 , 3 , 0 , D , 2 , 7 , I ] ready2   182
182 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , E , 2 , 7 , I ] mored7   183
183 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 2 , 7 , I ] xmit7    184
184 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 3 , 7 , I ] rcv3     185
185 [ 0 , 0 , 1 , 0 , 0 , 0 , 3 , 0 , D , 3 , 7 , I ] ready3   186
186 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , E , 3 , 7 , I ] mored7   187
187 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 3 , 7 , I ] xmit7    188
188 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 4 , 7 , I ] rcv4     189
189 [ 0 , 0 , 0 , 1 , 0 , 0 , 3 , 0 , D , 4 , 7 , I ] ready4   190
190 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , E , 4 , 7 , I ] mored7   191
191 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 4 , 7 , I ] xmit7    192
192 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 5 , 7 , I ] rcv5     193
193 [ 0 , 0 , 0 , 0 , 1 , 0 , 3 , 0 , D , 5 , 7 , I ] ready5   194
194 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , E , 5 , 7 , I ] mored7   195
195 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 5 , 7 , I ] xmit7    196
196 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 6 , 7 , I ] rcv6     197
197 [ 0 , 0 , 0 , 0 , 0 , 1 , 3 , 0 , D , 6 , 7 , I ] ready6   198
198 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , E , 6 , 7 , I ] mored7   199
199 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 6 , 7 , I ] xmit7    200
200 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , D , 8 , 7 , I ] rcv8     201
201 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 1 , D , 8 , 7 , I ] ready8   202
202 [ 0 , 0 , 0 , 0 , 0 , 0 , 3 , 0 , E , 8 , 7 , I ] pass_tk7 203
203 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 8 , 7 , E ] get_tk8  204
204 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 8 , 7 , E ] xmit8    205
205 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 1 , 8 , I ] rcv1     206
206 [ 1 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 1 , 8 , I ] ready1   207
207 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , E , 1 , 8 , I ] mored8   208
208 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 1 , 8 , I ] xmit8    209
209 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 2 , 8 , I ] rcv2     210
210 [ 0 , 1 , 0 , 0 , 0 , 0 , 0 , 3 , D , 2 , 8 , I ] ready2   211
211 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , E , 2 , 8 , I ] mored8   212
212 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 2 , 8 , I ] xmit8    213
213 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 3 , 8 , I ] rcv3     214
214 [ 0 , 0 , 1 , 0 , 0 , 0 , 0 , 3 , D , 3 , 8 , I ] ready3   215
215 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , E , 3 , 8 , I ] mored8   216
216 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 3 , 8 , I ] xmit8    217
217 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 4 , 8 , I ] rcv4     218
218 [ 0 , 0 , 0 , 1 , 0 , 0 , 0 , 3 , D , 4 , 8 , I ] ready4   219
219 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , E , 4 , 8 , I ] mored8   220
220 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 4 , 8 , I ] xmit8    221
221 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 5 , 8 , I ] rcv5     222
222 [ 0 , 0 , 0 , 0 , 1 , 0 , 0 , 3 , D , 5 , 8 , I ] ready5   223
223 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , E , 5 , 8 , I ] mored8   224
224 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 5 , 8 , I ] xmit8    225
225 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 6 , 8 , I ] rcv6     226
226 [ 0 , 0 , 0 , 0 , 0 , 1 , 0 , 3 , D , 6 , 8 , I ] ready6   227
227 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , E , 6 , 8 , I ] mored8   228
228 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 6 , 8 , I ] xmit8    229
229 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , D , 7 , 8 , I ] rcv7     230
230 [ 0 , 0 , 0 , 0 , 0 , 0 , 1 , 3 , D , 7 , 8 , I ] ready7   231
231 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 3 , E , 7 , 8 , I ] pass_tk8 232
232 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 1 , 8 , E ] get_tk1  233
233 [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 1 , 8 , E ] pass1    234
234 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 2 , 1 , E ] get_tk2  235
235 [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 2 , 1 , E ] pass2    236
236 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 3 , 2 , E ] get_tk3  237
237 [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 3 , 2 , E ] pass3    238
238 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 4 , 3 , E ] get_tk4  239
239 [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 4 , 3 , E ] pass4    240
240 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 5 , 4 , E ] get_tk5  241
241 [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 5 , 4 , E ] pass5    242
242 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 6 , 5 , E ] get_tk6  243
243 [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 6 , 5 , E ] pass6    244
244 [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 7 , 6 , E ] get_tk7  245
245 [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 7 , 6 , E ] pass7    246
```

```
246  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 8 , 7 , E ] get_tk8  247
247  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 2 , E , 8 , 7 , E ] pass8    248
248  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 1 , 8 , E ] get_tk1  249
249  [ 2 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , E , 1 , 8 , E ] pass1    250
250  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 2 , 1 , E ] get_tk2  251
251  [ 0 , 2 , 0 , 0 , 0 , 0 , 0 , 0 , E , 2 , 1 , E ] pass2    252
252  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 3 , 2 , E ] get_tk3  253
253  [ 0 , 0 , 2 , 0 , 0 , 0 , 0 , 0 , E , 3 , 2 , E ] pass3    254
254  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 4 , 3 , E ] get_tk4  255
255  [ 0 , 0 , 0 , 2 , 0 , 0 , 0 , 0 , E , 4 , 3 , E ] pass4    256
256  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 5 , 4 , E ] get_tk5  257
257  [ 0 , 0 , 0 , 0 , 2 , 0 , 0 , 0 , E , 5 , 4 , E ] pass5    258
258  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 6 , 5 , E ] get_tk6  259
259  [ 0 , 0 , 0 , 0 , 0 , 2 , 0 , 0 , E , 6 , 5 , E ] pass6    260
260  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 7 , 6 , E ] get_tk7  261
261  [ 0 , 0 , 0 , 0 , 0 , 0 , 2 , 0 , E , 7 , 6 , E ] pass7    262
262  [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , 0 , T , 8 , 7 , E ] get_tk8  247

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
---------------------------------------------------
Number of states generated :263
Number of states analyzed :263
Number of deadlocks : 0


                     UNEXECUTED TRANSITIONS
                         *****NONE*****
```

LIST OF REFERENCES

1. Lundy, G. M., and Miller, R. E., "Specification and Analysis of a Data Transfer Protocol Using Systems of Communicating Machines," *Distributed Computing*, Springer-Verlag, December 1991.

2. Lundy, G. M., and Miller, R. E., "Specification and Analysis of a General Data Transfer Protocol," Tech Rep GIT-88/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 1988.

3. Lundy, G. M., and Akyildiz I. F., "A Formal Model of the FDDI Network Protocol," *Europa Proceedings of the EFOC/LAN'91*, pp. 201-205, London, 1991.

4. Lundy, G. M., "Specification and Analysis of the Token Bus Protocol Using Systems of Communicating Machines," IEEE Systems Design and Networks Conference, Santa Clara, CA, 1990.

5. Lundy, G. M., and Luqi, "Specification of Token Ring Protocol Using Systems of Communicating Machines, "IEEE Systems Design and Networks Conference, Santa Clara, CA, 1989.

6. Lundy, G. M., and Miller, R. E., "Analyzing a CSMA/CD Protocol Through a Systems of Communicating Machines Specification (submitted for publication).

7. Raiche, C., "Specification and Analysis of The Token Ring Protocol," M. S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 1989.

8. Rothlisberger, M. J., "Automated Tools for Validating Network Protocols," M. S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, September 1992.

9. Peng, Wuxu and Puroshothaman, S., "Data Flow Analysis of Communicating Finite State Machines," *ACM Transactions on Programming Languages and Systems,* Vol.13, No. 3, July 1991.

10. Rudin, H., "An Informal Overview of Formal Protocol Specification," IFIP TC 6th International Conference on Information Network and Data Communication, Ronneby Brunn, Sweden, 11-14 May 1986.

11. Vuong, S. T., and Cowan, D. D., "Reachability Analysis of Protocols with FIFO Channels," ACM SIGCOMM, University of Texas at Austin, March 8-9 1983.

12. Gouda, M. G., “An Example for Constructing Communicating Machines by Stepwise Refinement,” *Proc. 3rd IFIP WG 6.1 Int. Workshop on Protocol Specification, Testing, and Verification,* North-Holland Publ., 1983.

13. United States, Department of Defense, “Reference Manual for the Ada Programming Language,” ANSI/MIL-STD-1815A-1983.

14. Lundy G. M., “Modeling and Analysis of Data Link Protocols,” TN86-499.1, Telecommunications Research Laboratory, GTE Laboratories, 40 Sylvan Road, Waltham, MA, January 1986.

15. Charbonneau, L. J., “Specification and Analysis of The Token Bus Protocol,” M. S. Thesis, Department of Computer Science, Naval Postgraduate School, Monterey, CA, 1990.

16. Holzmann, Gerard J., “Design and Validation of Computer Protocols,” Prentice Hall Publishing Co., 1991.

17. Aggarwal S., Barbara D., and Meth K. Z., “SPANNER: A Tool for the Specification, Analysis, and Evolution of Protocols,” IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, December 1987.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library, Code 052 Naval Postgraduate School Monterey, CA 93943	2
3.	Chairman, Code 37 CS Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr. G. M. Lundy, Code CS/Ln Assistant Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
5.	Dr. Man-Tak Shing, Code CS/Sh Associate Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Dr. Mohamed Gouda Department of Computer Science University of Texas at Austin Austin, TX 78712	1
7.	Dr. Raymond E. Miller Department of Computer Science A. V. Williams Bldg. University of Maryland College Park, MD 20742	1
8.	Dr. Krishan Sabnani AT&T Bell Labs Room 2C-218 Murray Hill, NJ 07974	1

9. Deniz Kuvvetleri Komutanligi 1
Personel Daire Baskanligi
Bakanliklar, Ankara / TURKEY

10. Golcuk Tersanesi Komutanligi 1
Golcuk, Kocaeli / TURKEY

11. Deniz Harp Okulu Komutanligi 1
81704 Tuzla, Istanbul / TURKEY

12. Taskizak Tersanesi Komutanligi 1
Kasimpasa, Istanbul / TURKEY

13. LTJG Zeki Bulent Bulbul 1
Merkez Bankasi Evleri
Ozgurler Sok. No. 9
Kalaba, Ankara / TURKEY